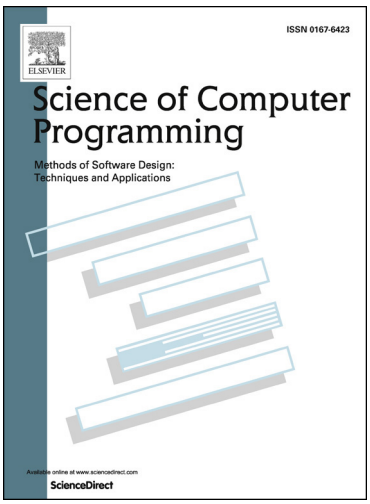
Accepted Manuscript



Formal development of multi-agent systems using MAZE

Qin Li, Graeme Smith

PII: S0167-6423(16)30017-X
DOI: http://dx.doi.org/10.1016/j.scico.2016.04.008
Reference: SCICO 2007

To appear in: *Science of Computer Programming*

Received date: 30 January 2015
Revised date: 25 April 2016
Accepted date: 29 April 2016

Please cite this article in press as: Q. Li, G. Smith, Formal development of multi-agent systems using MAZE, *Sci. Comput. Program.* (2016), http://dx.doi.org/10.1016/j.scico.2016.04.008

Highlights

- MAZE is a formal specification and development notation for multi-agent systems.
- MAZE supports three distinct levels of abstraction and action refinement to move between these levels.
- MAZE includes syntactic conventions for abstractly specifying low-level communication and timing constraints.
- Practical simulation rules for proving action refinement in MAZE are introduced.
- MAZE is applied to a non-trivial case study: a swarm robotic algorithm for self-assembly.

Formal development of multi-agent systems using MAZE

Qin Li[a,*], Graeme Smith[b]

[a]*Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, China.*
[b]*School of Information Technology and Electrical Engineering, The University of Queensland, Australia*

Abstract

MAZE is an extension of the Object-Z specification language supporting the specification and development of multi-agent systems (MAS). Following recommendations from the agent-oriented software engineering community, it supports three distinct levels of abstraction: (i) the *macro* level which focusses on the system's overall, global behaviour, independently of how the agents of the system operate and interact, (ii) the *meso* level which focusses on agent interactions, and (iii) the *micro* level which focusses on the operation of individual agents. Object-Z's high-level support for component-based specification, which is well suited to modelling MAS, is complemented in MAZE with support for action refinement to facilitate the top-down development process from the macro to micro level, and with a number of syntactic conventions aimed at abstractly specifying the low-level mechanisms required for dealing with asynchronous communication and timing constraints at the micro level. The latter are shorthands for existing Object-Z notation and so require no redefinition of Object-Z's semantics. In this paper, we provide an overview of MAZE and illustrate its use on a non-trivial case study: a swarm robotic algorithm for self-assembly.

1. Introduction

A *multi-agent system* (*MAS*) is a system comprising a number of interacting, *autonomous agents*, i.e., components which can initiate actions without external control. Our notion of an agent includes not only "intelligent" agents [1, 2], but also components which autonomously follow simple protocols such as the sensors in a self-organising sensor network [3], or the nodes of an *ad-hoc* mobile network which continually adapt their routing patterns to the current network topology [4].

The components (agents) in a MAS have the following characteristics [1].

- **Autonomy**: The agents in a MAS are independent, self-aware and autonomous. An agent has its own goals or follows its own behavioural rules without external intervention.

*Corresponding author.
Email addresses: qli@sei.ecnu.edu.cn (Qin Li), smith@itee.uq.edu.au (Graeme Smith)

- **Locality**: Every agent has only a local view on its environment. It is inefficient or even impossible for an agent to get a global view of the whole system.
- **Decentralisation**: There is no centralised control throughout the system. The agents follow distributed protocols to interact and share their knowledge and experience with each other.

The above characteristics result in so-called "self-organising" behaviours of the system [5]. Complex system-level behaviour may emerge even when the rules that individual agents follow are simple. Engineering the behaviour of individual agents and their interaction protocols to realise the required system-level behaviours is a promising but challenging topic in the MAS field.

Zambonelli and Omicini [6] argue that the disciplined engineering of MAS should proceed at three distinct levels of abstraction.

1. At the *macro* level, the engineer is concerned with the overall system functionality, ignoring the operation and interaction of its agents.
2. At the *meso* level, the engineer considers potential agent interactions and interaction paradigms that will lead to the desired system functionality.
3. At the *micro* level, the engineer is concerned with the operation of individual agents, choosing an implementation that results in the required meso-level interactions.

A correct design of MAS should keep the three levels consistent so that the systematic functionality can be achieved by the local behaviours and interactions of the agents.

We have adopted this three-level approach in the formal development of multi-agent systems [7, 8, 9], leading to the development of an extension to Object-Z [10] called MAZE [11]. The extension supports action refinement [12] as a means of developing a specification from the macro level, where operations are coarse-grained, through to the micro level, where the granularity of operations is usually much finer. In the micro-level specification, the above characteristics of agents are formalised by adopting a special framework using Object-Z notation. Simulation rules for checking action refinement provide proof obligations that designers should satisfy at each development step. We show through our case study that they are useful for detecting design problems as well as selecting solutions. The extension also involves a number of syntactic conventions, aimed at facilitating the specification of inter-agent communication mechanisms and associated timing constraints at the micro level. The syntactic conventions are analogous to those used in Z for modelling sequential systems [13]. That is, they are merely a shorthand for what could otherwise be expressed using more basic syntax. For this reason, they do not require an extension to the existing semantics of Object-Z.

In this paper, we extend the work in [11] by providing a more practical verification method for action refinement in MAZE and proving it sound with respect to a high-level, trace-based definition of action refinement. We also apply MAZE to a non-trivial case study: a swarm robotic self-assembly algorithm based on the work of Støy and Nagpal [14, 15]. We begin by introducing our case study in Section 2. In Section 3, we introduce MAZE and our proof method for action refinement in MAZE along with the proof of its soundness. In Section 4, we illustrate the use of MAZE for incrementally developing specifications of MAS from the macro to the micro level including the use of syntactic conventions for modelling individual agents and their interactions at the micro-level. Related work is discussed in Section 5 before we conclude the paper in Section 6.

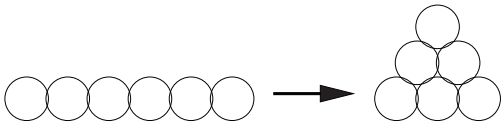


Figure 1: Two-dimensional representation of a connected mass of robots forming a desired shape.

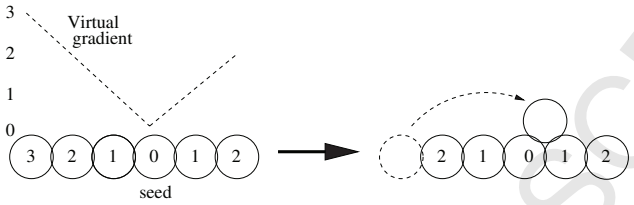


Figure 2: Robots follow a gradient established by the seed robot to reach a position in the target shape.

2. A gradient-based approach to self-assembly

Self-assembly algorithms allow a swarm of robots to autonomously form a shape or pattern appropriate for a given task. Many different algorithms have been devised for the process of *morphogenesis*, i.e., "growing" the pattern or shape from an unordered swarm of robots [16, 17, 15, 18, 19]. In this section, we describe the approach of Støy and Nagpal [14, 15]. Their algorithm utilises virtual gradients created and propagated by the robots in the swarm to recruit other robots in order for the mass to assemble into a required 3-dimensional shape. The self-assembly algorithm we verify in the paper is based loosely on this work; the differences are discussed below.

2.1. Støy and Nagpal's algorithm

The robots, referred to as *modules* by Støy and Nagpal, form a connected mass. Each robot is capable of local communication with immediate neighbours only, i.e., those with which they have physical contact. They are also capable of movement over or around their neighbours to form a required shape (see Figure 1).

Initially, one robot, called the *seed*, is provided with a representation of the shape to be formed (which we will refer to as the *target*). The seed assumes a particular position in this target, and then *recruits* other atoms to join the target. This is achieved by setting up a *recruitment gradient* to attract other robots to neighbouring positions in the target. The recruitment gradient is a virtual gradient, or slope, represented by integer values stored by each of the robots (see left-hand side of Figure 2). It is set up by the seed robot storing an integer value, say 0, then incrementing that value by 1 and broadcasting the result to each of its neighbours. These store the received value, increment it by 1 and then broadcast it to each of their neighbours. The result is that each robot stores its distance from the seed (see left-hand side of Figure 2).

A robot follows a gradient by moving from a position next to a robot with gradient value n to a position next to another robot with gradient value $n-1$. By following a gradient in this way it eventually reaches the seed (see right-hand side of Figure 2). The seed passes such a robot the target representation and it takes on one of the vacant target positions. After this, it acts like a seed to recruit any neighbours it requires. Note that

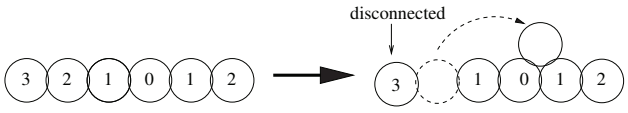


Figure 3: Robots should not become disconnected from the swarm as a consequence of other robots moving.

a seed robot does **not** behave as a central controller of the system, which would violate the notion of **decentralisation**. It only interacts with its immediate neighbours.

For the algorithm to work, Støy and Nagpal identify two constraints. The first of these is that the robots must remain connected. If one or more robots becomes disconnected from the rest, they can no longer receive messages (see Figure 3). Hence, an additional virtual gradient, called the *connection gradient*, is initially established by the seed, and a robot can only move if its distance from the seed is greater than or equal to those of its neighbours, i.e., if it is not required to connect its neighbours to the mass.

The second constraint is that the target shape must have enough spaces in it to allow robots to move to any required position. This is ensured by requiring that target shapes conform to a particular porous "scaffold" structure. Other than conforming to such a structure there are no constraints on the target shape except that it must be a connected mass of robots. Various methods for efficiently representing the target within a robot have been proposed by Støy and Nagpal; in this paper we abstract from such details.

2.2. Modifications to the algorithm

The algorithm we develop in this paper differs from that of Støy and Nagpal in two aspects. Firstly, they do not discuss the issue of multiple, intersecting gradients. This can occur due to more than one seed creating a gradient (see Figure 4), or when a single gradient returns to a robot due to a loop in the swarm structure. To deal with this, we require a robot which already has a recruitment gradient value to ignore any values that it receives which would result in increasing this gradient value. This results in robots storing a value representing the *shortest* distance to a seed of a recruitment gradient (e.g., the robot indicated by the arrow in Figure 4 joins and propagates the gradient of seed 1).

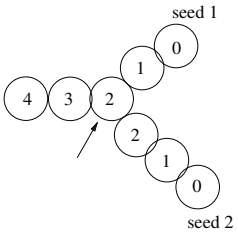


Figure 4: Robots at the intersection of two gradients form part of the gradient of the nearest seed.

To ensure progress under this approach, we require also that, once a robot has all the neighbours it needs, the gradient leading to it is 'cancelled' (thus allowing other gradients to propagate). This proposal will be specified, and verified to work, using our refinement-based approach, in Section 4.

The second change to the algorithm of Støy and Nagpal is that we do not have a connection gradient. Instead, we use the recruitment gradients to determine when an atom is closer than its neighbours to a seed atom in the target. If it is closer (such as the leftmost atom with gradient value 2 in Figure 3), then that atom should not move since its neighbours may rely on it staying in position to remain connected to the mass.

3. MAZE

3.1. Macro- and meso-level specification

A macro- or meso-level specification in MAZE is captured by the class construct of Object-Z [10]. It encapsulates a collection of type and constant definitions, a schema describing the specification's state space (in terms of variables and an invariant), a schema defining the initial state, and a set of operations defining possible agent actions. A macro-level specification has the form:

System

declarations of types $\vec{t}$ *and constants* $\vec{c}$

declarations of state variables $\vec{v}$

$inv(\vec{c}, \vec{v})$

INIT

$init(\vec{c}, \vec{v})$

Act_1

$\vdots$

Act_n

Each action in MAZE is defined with an Object-Z operation schema of the following form:

Act_i

$\Delta(\vec{u})$

declarations of local variables $\vec{x}$

$body(\vec{c}, \vec{v}, \vec{x}, \vec{v}')$

where $\vec{u}$ is the subset of the state variables $\vec{v}$ whose values can be changed by the operation (all other variable values remain unchanged), and $\vec{v}'$ denotes the post-state values of the variables $\vec{v}$. Semantically, Object-Z operations are *guarded*. When the predicate $body(\vec{c}, \vec{v}, \vec{x}, \vec{v}')$ cannot be satisfied then the operation cannot occur. This is in contrast to Z operations which can occur at any time but have undefined behaviour when their predicate cannot be satisfied [13].

Often the system is modelled at the macro level by a single action which reaches the desired state. Termination in the desired state is guaranteed by making the guard of the action evaluate to false in the desired state. In general, when there is more than one action or the action can be executed more than once, we need to prove that the specification reaches a state where no action is enabled and that the negation of the disjunction of the actions' guards implies the desired state.

This approach will not reflect the system behaviour in the case when it is non-terminating (since the execution terminates when the desired state is reached). However, it is sufficient for verifying, through the successive refinement of the action to a sequence of finer-grained actions at the meso and micro level, that a particular MAS design produces a desired goal under certain conditions (captured by the initial state schema). This will be demonstrated in Section 4.

3.2. Action refinement

Macro-level specifications abstract from interactions between agents, focussing instead on the outcomes of those interactions. The goal at the meso level of development is to decompose the abstract actions of the macro level to actions representing agent interactions. The latter are still in terms of the global state of the system and act as a bridge between the macro level and micro level where individual agent behaviours are specified.

Adding the interactions as we develop the specification to the meso level, and ultimately the micro level, requires the addition of further actions, e.g., to model the sending and receiving of messages. Previous work in this area [9] has shown it is desirable to allow action guards to be strengthened; refinement preserving overall enabledness, rather than enabledness of each action (as required by the notions of *weak* and *non-atomic refinement* for Object-Z [20]). We therefore base our approach on the simulation rules for action refinement of action systems by Back and von Wright [12]. Our decision to use Object-Z, rather than action systems or Event-B [21] (which uses a form of action refinement based on that of action systems), is due to its expressive power and its high-level support for component-based specification. Below we consider the forward simulation rules for action refinement, and adapt them to Object-Z. The backwards simulation rules could be similarly adapted.

Action refinement allows us to develop several concrete actions whose sequential composition simulates an abstract action. When action refinement is considered between two action systems, we only consider their observable behaviours. A pair of mappings $f : \Sigma_A \to \Sigma_E$ and $g : \Sigma_C \to \Sigma_F$ map the state of the abstract and concrete system, respectively, to the state of their observable variables. A relation $h : \Sigma_E \leftrightarrow \Sigma_F$ from the observable abstract state space to the observable concrete state space is defined to indicate which states are considered equivalent in both systems. In many cases, h is an identity relation and all variables of the abstract system are observable (i.e., $\Sigma_A = \Sigma_E$). A transition is called a *stuttering transition* of a system if its pre-state and post-state are mapped to the same observable state according to the relevant mapping f or g; otherwise, the transition is called a *change transition*.

Given a computation c of a system, we can obtain an observable trace by removing any stuttering transitions. Let A be the abstract system and C be the concrete system. We use the notation $tr(A, f)$ and $tr(C, g)$ to denote the set of observable traces in A and C respectively. In general, we can define a retrieve relation $R \mathrel{\widehat{=}} f;\ h;\ g^{-1}$ relating the abstract state space to concrete state space, i.e., $R : \Sigma_A \leftrightarrow \Sigma_C$.

Definition 1. (Action refinement) Let C and A be action systems with mappings f, g and relation h. $R = f;\ h;\ g^{-1}$ is a retrieve relation. We say C is a refinement of A with respect to R, denoted by $C \sqsupseteq_R A$, if and only if for any finite trace t in $tr(C, g)$, there exists a finite trace s in $tr(A, f)$ such that $\#s = \#t$ and $\forall\, i : 1..\#t \bullet (s[i], t[i]) \in h$; and for any infinite trace t in $tr(C, g)$, there exists an infinite trace s in $tr(A, f)$ such that $\forall\, i : \mathbb{N} \bullet (s[i], t[i]) \in h$. $\diamond$

The definition of action refinement implies that

1. for any pair $(t[i], t[i+1])$ in the concrete system from and to observable concrete states, there is a pair $(s[i], s[i+1])$ in the abstract system from and to observable abstract states that are related by relation h, which implies their corresponding abstract and concrete state are related by relation R;
2. if an abstract observable trace can be extended with an observable abstract state, the concrete trace can be extended with a related observable concrete state that is related to the abstract one by relation h, which implies their corresponding abstract and concrete state are related by relation R.

For a given Object-Z specification, the change and stuttering transitions are particular

occurrences of the specification's operations, i.e., particular pre-state/post-state pairs that satisfy an operation's predicate. A single operation can have some occurrences which are change transitions, and some which are stuttering transitions, depending on whether or not the post-state of the operation is related to the same observable state as the pre-state. Examples of this will be given when we return to our case study in Section 4.

Proving action refinement using Definition 1 is not practical due to the need to perform checks on traces. Hence we introduce a simulation-based approach which simplifies the refinement checking to checks on actions. Although the conditions (inspired by those of Event-B [21]) are not complete, and slightly stronger than Definition 1, they are much simpler to check.

Definition 2. (Forward Simulation) Let A be an Object-Z class with state schema $AState$, initial state schema $AInit$, and operation occurrences partitioned into change transitions $AChange_0, \ldots, AChange_n$ and stuttering transitions $AStutt_0, \ldots, AStutt_m$ for some $n, m : \mathbb{N}$. Similarly, let C be an Object-Z class with state schema $CState$, initial state schema $CInit$, and operation occurrences partitioned into change transitions $CChange_0, \ldots, CChange_l$ and stuttering transitions $CStutt_0, \ldots, CStutt_k$ for some $l, k : \mathbb{N}$. A is refined by C when there exists a retrieve relation $R : \Sigma_A \leftrightarrow \Sigma_C$ (modelled by a Z schema as in [20]) which relates the states of A to those of C such that the following hold. (Schemas are used below as declarations and predicates as in Z [13]. The Z notation pre $Action$ returns the guard of an Object-Z operation $Action$.)

Initialisation: Initialisation in C simulates initialisation in A.

$$\forall\, CState \bullet CInit \Rightarrow (\exists\, AState \bullet AInit \wedge R)$$

Action Simulation: Any change transition $CChange_c$ in C simulates some change transition $AChange_a$ in A; any stuttering transition $CStutt_c$ in C simulates the identity transition in A (denoted ID_A).

$$\forall\, AState, CState, CState' \bullet R \wedge CChange_c \Rightarrow (\exists\, AState' \bullet AChange_a \wedge R')$$

$$\forall\, AState, CState, CState' \bullet R \wedge CStutt_c \Rightarrow (\exists\, AState' \bullet ID_A \wedge R')$$

Termination: Any terminating state in C is related only to terminating states in A.

$$\begin{array}{l} \forall\, AState, CState \bullet \\ \quad R \wedge \neg\ \mathrm{pre}(CChange_0 \vee \ldots \vee CChange_l \vee CStutt) \Rightarrow \\ \qquad \neg\ \mathrm{pre}(AChange_0 \vee \ldots \vee AChange_n \vee AStutt) \end{array}$$

where $AStutt = (AStutt_0 \vee \ldots \vee AStutt_m)$ and $CStutt = (CStutt_0 \vee \ldots \vee CStutt_k)$.

Stuttering Convergence: Any state in C from which infinite stuttering is possible is related only to states in A from which infinite stuttering is possible.

$$\begin{array}{l} \forall\, AState, CState \bullet \\ \quad R \wedge (\forall\, i : \mathbb{N} \bullet \exists\, CState' \bullet CStutt^i \wedge (\mathrm{pre}\, CStutt)') \Rightarrow \\ \qquad (\forall\, j : \mathbb{N} \bullet \exists\, AState' \bullet AStutt^j \wedge (\mathrm{pre}\, AStutt)') \end{array}$$

where T^i means sequentially performing transition T for i times. $\diamond$

To prove the above simulation rules are sound, we need to show that they are sufficient to imply action refinement as defined in Definition 1.

Soundness Proof: Let $R = f;\ h;\ g^{-1}$. For any computation c of a concrete system C with observable trace t belonging to $tr(C, g)$, we need to construct a trace s belonging to $tr(A, f)$ such that if t is finite, $\#s = \#t$ and $\forall\, i : 1..\#t \bullet (s[i], t[i]) \in h$; and if t is infinite, s is infinite and $\forall\, i : \mathbb{N} \bullet (s[i], t[i]) \in h$. The constructive proof is as follows.

(1) According to the Initialisation rule, there exists an abstract initial state (say σ_1) which is related to $c[1]$ by R, i.e., $(f(\sigma_1), t[1]) \in h$. We let $s = \langle f(\sigma_1) \rangle$ for some such σ_1.

(2) For each further state $c[i]$ of c in turn, we proceed as follows. The transition $(c[i-1], c[i])$ is either a change transition or a stuttering transition.

(2.1) If it is a change transition, according to the Action Simulation rule, there is an abstract state σ_i such that the pair (σ_{i-1}, σ_i) belongs to a change transition of A and $(f(\sigma_i), t[i]) \in h$. We append $f(\sigma_i)$ to s.

(2.2) If it is a stuttering transition, we have $g(c[i-1]) = g(c[i])$ and $c[i]$ is removed in t. According to the Action Simulation rule, σ_{i-1} is related to $c[i]$ by R.

We then have $\#s = \#t$ and for $i \in 1..\#t$, $(s[i], t[i]) \in h$ if t is finite, and s is infinite and for $i \in \mathbb{N}$, $(s[i], t[i]) \in h$ if t is infinite. Next we need to show that s cannot be extended beyond the observable transitions of t when t is finite.

(4) According to the Termination rule, if c ends with a terminating state, all abstract states related by R are terminating states in A.

(5) The Stuttering Convergence rule guarantees that if the concrete computation c diverges, i.e., c is infinite while t is finite ending in a divergent state from which infinite stuttering is possible, the abstract trace s also ends in a divergent state. In the case where the abstract system has no stuttering transitions, the Stuttering Convergence rule guarantees that c does not diverge. □

In the case where all transitions in the abstract system are change transitions, the stuttering convergence condition can be simplified to require that the execution of stuttering transitions in the concrete system converges, i.e., they can be only executed a finite number of times. It can be expressed as

$$\forall\, CState \bullet \exists\, N : \mathbb{N} \bullet \forall\, CState' \bullet CStutt^N \Rightarrow \neg\ (\mathrm{pre}\ CStutt)' \qquad (*)$$

In general cases, it is sufficient to prove the above convergence condition by constructing a variant W whose value is a natural number or a finite set so that every stuttering transition in concrete system C decreases it, e.g.,

$$\forall\, CState, CState' \bullet CStutt \Rightarrow W' \subset W\ .$$

With the variant, the proof can be done by just considering the stuttering transitions once rather than checking their iterative executions. However, in more complex cases, constructing such a variant is non-trivial. In such cases, we have to analyse the behaviours of the stuttering transitions to prove condition $(*)$ is true.

3.3. Micro-level specification

At the micro level of development we produce a specification of the local behaviour of individual agents and implement their interactions specified at the meso level. MAZE supports the definition of agents in terms of their local variables and actions in an independent class structure like that used for the system at the macro and meso levels. The actions of each agent, when enabled, are implicitly able to occur at any time within the system specification, i.e., they are not controlled by their environment. This captures the notion of **autonomy** of agents.

In MAZE, every agent can only manage its local state. Any information other than its local state can be only accessed by an agent through interaction with reachable agents via asynchronous message-passing. This captures the notion of **locality**. The use of asynchronous communication is used to capture the notion of **decentralisation**. As there is no centralised control, messages may be sent to an agent at any time, including when it is busy with another message. Hence, messages need to be buffered (since allowing messages to be lost would greatly complicate the simple micro-level protocols we are trying to establish). In the implementation of a MAS, the buffering may be part of the communication medium, e.g., when agents are distributed over a network, or part of the agent, e.g., when communication is wireless and effectively synchronous between agents. The robot self-assembly case study of this paper is an example of the latter, where agent-to-agent communication occurs via direct connections between atoms which are in physical contact with each other.

The behaviours of the collection of agents (which are instances of the classes) and their interaction effects on their environment are captured further via a system specification. The system specification at the micro level differs from those at other levels in that it refers to the specification(s) of individual agents. The semantics, based on that of object instantiation in Object-Z [10], enables agent instances to be declared and the state of such instances to be accessed using the standard notation from object orientation, e.g., the notation $a.v$ denotes the constant or state variable v of agent a. We also allow local types of agent specifications to be accessed via dot notation, e.g., $A.T$ denotes the local type T of agent specification A. For each agent specification in MAZE, these local types include a type **message** defined as a Z free type. It defines the kinds of messages the agent can send and receive.

Although it is possible to specify asynchronous message passing in standard Object-Z, this can lead to specifications which are awkward to read due to the details of the particular system under development being intermingled with those of the underlying communication mechanisms. In MAZE, we separate these details by capturing the latter implicitly via a number of syntactic conventions.

Special syntax is introduced to specify a collection of interacting agents. $\mathbb{T}\, A$ defines a topology of agents of type A in terms of a finite function whose domain is the set of all agents in the topology and which maps each such agent to the agents to which it can send messages, i.e., $\mathbb{T}\, A = (A \twoheadrightarrow \mathbb{F}\, A)$. Note that there are no constraints on the function allowing uni-lateral sending of messages as well as agents which are isolated and unable to send or receive messages. Typically, constraints will be added in the specification to restrict the function as required.

The notation $\mathbb{T}\, A$ not only introduces a topology of agents of type A, but also implicitly introduces their initialisation (according to the initial state schema of A) and system

actions allowing any agent in the topology to perform any enabled action. These actions send messages to and receive messages from an implicit global buffer. The buffer is unordered allowing messages to be received by the agent in a different order to which they are sent. This may model the use of different routes through a communication medium such as the Internet, or the ability of an agent to prioritise messages in its internal buffer.

Finally, the system specification may include explicitly defined *system actions* that change the agents' environment and topology. Such actions may occur either independently (when the environment itself can change) or in response to an agent action. In the latter case we follow the name of the system action with a tag $< a : \mathrm{dom}\, t \bullet a.Act >$ (where t is an object topology). The tag declares an agent a belonging to the topology t, and an action of that agent A which occurs together with the system action, and not otherwise. The agent a declared by the tag may be referenced throughout the system action's definition.

Definition 3. (System specification) The following two specifications are semantically equivalent where $Act_1, \ldots, Act_n$ are the actions of agent specification A, $SysAct1$ is an example of an independently occurring system action, and $SysAct2$ an example of a system action which occurs together with an agent action. ([] is the Object-Z distributed choice operator which is semantically equivalent to an existential quantifier [10]). The left-hand specification is that which the specifier would write.

S

$t : \mathbb{T}\, A$

$SysAct_1$

details of $SysAct_1$

$SysAct_2 < a : \mathrm{dom}\, t \bullet a.Act_1 >$

details of $SysAct_2$

S

$t : A \twoheadrightarrow \mathbb{F}\, A$

$\mathbf{buffer} : \mathrm{bag}(A.\mathbf{message} \times A \times A)$

INIT

$\forall a : \mathrm{dom}\, t \bullet a.\mathit{INIT}$

$\mathbf{buffer} = [\![\,]\!]$

$SysAct_1$

details of $SysAct_1$

$SysAct_2 \mathrel{\widehat=} [] a : \mathrm{dom}\, t \bullet a.Act_1 \wedge [\mathit{details\ of}\ SysAct_2]$

$Act_2 \mathrel{\widehat=} [] a : \mathrm{dom}\, t \bullet a.Act_2$

$\vdots$

$Act_n \mathrel{\widehat=} [] a : \mathrm{dom}\, t \bullet a.Act_n$

where the implicit variable **buffer** models the unordered, global buffer as a bag (allowing an agent to send a message multiple times). Each element of the buffer is a tuple (m, a, b) where m is a message, a is the agent which sent m, and b is the agent to which the message has been sent. ◇

For specifying agent actions in MAZE, two message-related predicates are introduced: **send** for modelling messages being sent to the global buffer ($\mathbf{send}(m, a)$ models a message m being sent to agent a, and $\mathbf{send}(m)$ models a message being broadcast to all

connected agents according to the topology) and **receive** for modelling messages being received from the buffer (**receive**(m, a) models the receipt of a message m from agent a). Each action in an agent specification may have one or more **receive** predicates (as part of its guard) and one or more **send** predicates (as part of its postcondition).

A predicate **progress**(s, r) is also introduced for use in agent specifications. It is true when the system has progressed to a point where all messages in set s that have been sent by the agent and all messages in set r that have been sent to the agent have been received. This mechanism provides a way of abstracting from the use of timers and timing constraints required in an implementation of the MAS [22, 23] as illustrated below.

For agents of type A, the formal definitions of **send**, **receive** and **progress** are given in terms of the implicit variable **buffer** in the system specification as follows.

Definition 4. (Message passing) When we apply the agent action Act (using the notation $a.Act$) in a MAS specification with a topology of agents t, the **receive** and **send** predicates introduce an additional guard G and additional postcondition $\mathbf{buffer}' = (\mathbf{buffer} \uplus R) \uplus S$ to the action where

- $(m, b, a) \sqsubseteq \mathbf{buffer}$ is a conjunct of G if, and only if, Act includes **receive**(m, b).
- (m, b, a) is in R if, and only if, Act includes **receive**(m, b).
- (m, a, b) is in S if, and only if, $b \in t(a)$ and Act includes **send**(m, b) or **send**(m).

$\diamond$

Definition 5. (Progress) When we apply the agent operation Act (using $a.Act$), each predicate of the form **progress**(s, r) where s and r are of type $\mathbb{P}\,\mathbf{message}$ introduces an additional guard:

$$\forall\, b : A \bullet (\forall\, m : s \bullet (m, a, b) \notin \mathbf{buffer}) \wedge (\forall\, m : r \bullet (m, b, a) \notin \mathbf{buffer}) \qquad \diamond$$

To illustrate the use of these message-related predicates, imagine a system in which an agent $a : Agent$ broadcasts a request for assistance with a certain task. Those neighbouring agents that are able to assist respond to the broadcast message, and all others ignore it. Agent a needs to wait for all responses before it can proceed with dividing the task. Since it doesn't know how many responses it is waiting on, this would probably be implemented by waiting for a given time. In MAZE, we would abstract from this timing constraint using the **progress** predicate (see below).

The operation for sending a request would be specified with a **send** predicate with a single parameter ($request$: **message**). The operation for receiving and responding to a message would be specified with a **receive** and **send** predicate. In this case, the **send** predicate would have a second parameter to specify that the message is sent only to the agent which initiated the request. The operation for ignoring a request would have just a **receive** predicate. Of course, the operations would have additional predicates (elided below) relating to when they would send a request, and under what circumstances they would respond to or ignore a request they have received.

SendRequest

send(*request*)
...

Respond

$a : Agent$

receive(*request*, a)
...
send(*response*, a)

IgnoreRequest

$a : Agent$

receive(*request*, a)
...

The agent a which sent the request would also have an operation to receive responses, and another to divide the task once all responses have been received. The latter should only occur after all neighbouring agents have received the request, and all subsequent responses have been received by a. In a real system, this would involve a timing constraint based on maximum times for message transmission between connected agents, and maximum message processing and response times [22, 23]. In MAZE, we abstract from this using a **progress** predicate which does not allow the operation to occur while either *request* messages from a, or *response* messages to a, remain in the buffer, i.e., while such messages have not yet been received.

ReceiveResponse

$b : Agent$

receive(*response*, b)
...

DivideTask

progress(*request*, *response*)
...

4. Case study

In our earlier work [11], we illustrated MAZE on a simple leader-election protocol employed in the cluster-based routing protocol of Gerla et al. [4]. Here we apply it to a more significant case study: the development of Støy and Nagpal's self-assembly algorithm. The formal development of the self-assembly robotic system is organised into the three levels of development with several refinement steps at each level.

4.1. Macro level

The initial macro-level specification *System*1 has a constant *target* which is a finite set of positions in 3-dimensional space representing the desired configuration of the robots. A variable *atoms* represents the positions of the finite set of robots, referred to in this paper as 'atoms'. An invariant constrains the number of atoms to be the same as the number of positions in the target shape. We introduce the type *Position* without constraining it in any way. We could be more precise and define positions to consist, for example, of triples of real numbers. For our purposes, however, the more abstract definition suffices

*System*1

$[Position]$

$target : \mathbb{F}\, Position$

$atoms : \mathbb{F}\, Position$

$\#atoms = \#target$

INIT

$true$

Configure

$\Delta(atoms)$

$atoms \neq target \land atoms' = target$

The behaviour is modelled by a single action which reaches the desired goal. Initially, the positions of the atoms are not constrained. The action *Configure* captures the desired goal of the system: placement of an atom at each target position. It is enabled at most once after which the system's behaviour terminates.

In the more concrete macro-level specification *System*2, we distinguish those atoms that are fixed in a target position from those that are still able to move despite being in a target position: the former represented by a variable *placed*. There is no explicit restriction on *placed* initially, although it is implicitly restricted via the invariant which requires it to be a subset of $atoms \cap target$.

The action *Place* fixes a single atom in a target position, modelled by increasing the number of atoms in the set *placed*. It also allows the set *atoms* to change in any way that satisfies the invariant. The specification performs *Place* once for each atom that is not initially in *placed*. After this, the specification terminates with $placed = target$ and hence due to the invariant the desired state, $atoms = target$.

The mappings f and h for this refinement step are identity relations. The mapping g captures the fact that the atoms are only regarded as being observed when the target is achieved. To improve readability, we decorate the variables in both specifications with subscripts , i.e., v_1 denotes variable v from *System*1 and v_2 denotes v from *System*2, and write the observable variables using a sans serif font.

*System*2

$[Position]$

$target : \mathbb{F}\, Position$

$atoms : \mathbb{F}\, Position$
$placed : \mathbb{F}\, Position$

$\#atoms = \#target$
$placed \subseteq atoms \cap target$

Init

$true$

Place

$\Delta(placed)$

$placed \subset target$
$\exists\, p : target \setminus placed \bullet$
$\quad placed' = placed \cup \{p\}$

$f \mathrel{\widehat=} target_1 = \mathsf{target}_1 \land atoms_1 = \mathsf{atoms}_1$
$h \mathrel{\widehat=} \mathsf{target}_1 = \mathsf{target}_2 \land \mathsf{atoms}_1 = \mathsf{atoms}_2$
$g \mathrel{\widehat=} \mathsf{target}_2 = target_2 \land (placed_2 = target_2 \Rightarrow \mathsf{atoms}_2 = atoms_2)$

The refinement from *System*1 to *System*2 can be verified using the retrieve relation $R1 \mathrel{\widehat=} f;\ h;\ g^{-1}$ relating the states from *System*1 to states from $System_2$.

$$R1 \mathrel{\widehat=} target_1 = target_2 \land (placed_2 = target_2 \Rightarrow atoms_1 = atoms_2)$$

For brevity, we directly give the definition of the retrieve relation R for all subsequent refinement steps in the case study instead of f, g and h.

According to the retrieve relation $R1$, the transitions of *System*2 are divided into change and stuttering transitions. Every occurrence of *Configure* is a change transition in *System*1. For *System*2, only the occurrence of *Place* where the post state establishes $placed_2 = target_2$ is a change transition. Its other occurrences are stuttering transitions. In this case, the proof of the rules of Definition 2 is straightforward.

Initialisation. Trivially holds since $R1$ relates each state of *System*2 to a state of

*System*1, and *Init* of *System*1 is true.

Action Simulation. Holds since each occurrence of *Place* where $placed_2 = target_2$ simulates *Configure*, and each other occurrence of *Place* simulates the identity transition of *System*1.

Termination. Holds since *System*2 terminates only when $placed_2 = target_2$ which is related to the terminating state $atoms_1 = target_1$ of *System*1.

Stuttering Convergence. The stuttering occurrences of *Place* in *System*2 will terminate when all atoms are in target positions. To prove this we use the number of atoms not in a target position as a variant, i.e., $W = \#target - \#placed$.

4.2. Meso-level

The goal at the meso level of development is to decompose the abstract actions of the macro level to actions representing agent interactions. The latter are still in terms of the global state of the system and act as a bridge between the macro level and micro level where individual agent behaviours are specified.

We begin developing the meso-level specification of our case study by decomposing action *Place* from *System*2. The strategy for placing atoms in our target implementation is to have atoms which are already placed recruit others. Each *Place* action could therefore be decomposed into a sequence of two concrete actions: the first corresponding to the recruitment, and the second to the recruited atom moving. This is specified in the class *System*3. To specify that only atoms with vacant neighbouring positions send recruitment messages, we introduce a constant $nb : Position \leftrightarrow Position$ denoting the neighbour relation between positions. We assume that the target is fully connected, i.e., $\forall p, q : target \bullet (p, q) \in nb^*$ where nb^* means the transitive closure of relation nb. To ensure an atom moves only when it is recruited, we introduce a variable $recruiters : \mathbb{F}\, Position$ to denote those atoms that have recruited others.

*System*3

$[Position]$

$target : \mathbb{F}\, Position$
$nb : Position \leftrightarrow Position$

$\forall p, q : target \bullet (p, q) \in nb^*$

$atoms : \mathbb{F}\, Position$
$placed : \mathbb{F}\, Position$
$recruiters : \mathbb{F}\, Position$

$\#atoms = \#target$
$placed \subseteq atoms \cap target$
$recruiters \subseteq placed$

Init

$recruiters = \varnothing$

$\vdots$

The action *Place* is then replaced by two concrete actions *Recruit* and *Move* defined as follows.

Recruit	*Move*
$\Delta(recruiters)$	$\Delta(atoms, placed)$
$p, q : Position$	$p, q : Position$
$(p, q) \in nb$	$(p, q) \in nb$
$p \in placed \setminus recruiters$	$p \in recruiters$
$q \in target \setminus placed$	$q \in target \setminus placed$
$recruiters' = recruiters \cup \{p\}$	$placed' = placed \cup \{q\}$

The action *Recruit* corresponds to an atom at position p recruiting an atom for a vacant neighbouring position q. The action *Move* corresponds to an atom moving to a vacant position neighbouring a 'recruiter' atom. Note that the set *atoms* is changed implicitly by this action.

In order to ensure that the above design of the system is correct, we need to check that *System*3 refines *System*2. Let $R2$ be the retrieve relation between the states of *System*2 and *System*3. We consider every variable in *System*2 to be mapped to the variable with the same name in *System*3.

$$R2 \mathrel{\widehat{=}} atoms_2 = atoms_3 \wedge target_2 = target_3 \wedge placed_2 = placed_3$$

Since $R2$ equates the variables *atoms*, *target* and *placed* of *System*2 and *System*3, these variables are linked via the mappings f, g and h, and hence are observable. Therefore, every occurrence of *Place* (since it changes variable *placed*) is a change transition. Similarly, every occurrence of *Move* (since it changes *placed* and possibly *atoms*) is a change transition. Occurrences of *Recruit* change only the variable *recruiters* which is non-observable. Hence, all occurrences of *Recruit* are stuttering transitions.

Initialisation. Holds since $recruiters = \varnothing$ implies *true* with the retrieve relation $R2$.

Action Simulation. We can easily prove that *Move* simulates *Place* (it has the same postcondition and a stronger guard). It is also trivial to prove that stuttering action *Recruit* simulates ID_{System_2} since it only changes the new variable *recruiters*.

Termination. The termination condition of *System*2 is $placed = target$. To satisfy this rule, the termination condition of *System*3 should not be stronger. *System*3 will terminate when both *Recruit* and *Move* are not enabled. According to their definitions and the invariant $recruiters \subseteq placed$, the termination condition of *System*3 holds when either

(a) there is no $p \in placed$, or

(b) there is such a p, and there is no $q \in target \setminus placed$ such that $(p, q) \in nb$.

In case (b), since *target* is fully connected and $placed \subseteq target$, we can deduce that $placed = target$. Hence, $\neg\, pre(Recruit \vee Move)$ is $placed = \varnothing \vee placed = target$. However, $\neg\, pre(Place)$ is $placed = target$ and so, with $R2$ as the retrieve relation, the termination condition does **not** hold.

This is a typical example where checking simulation rules can help in detecting design problems during development. It tells the designer that something needs to change in the concrete specification. One possible solution is to weaken the guard of *Recruit* so that it is enabled even when $placed = \varnothing$. However, this does not correspond to the design we are aiming at where only atoms fixed in target places recruit other atoms. Similarly, it does

not make sense to weaken the guard of *Move* to allow it to occur when $placed = \varnothing$. The other possibility is to strengthen the invariant of the concrete specification to exclude states where $placed = \varnothing$. This can be done either explicitly, or implicitly by strengthening the initial state to include $placed \neq \varnothing$ (since elements are never removed from *placed* by any action). The latter solution corresponds to the algorithm of Støy and Nagpal where a single (seed) atom is placed initially. This atom is the one that starts the self-assembly process. We modify the *Init* of *System*3 to capture this approach.

Init
$placed \neq \varnothing$
$recruiters = \varnothing$

It is trivial to check that this modification does not affect the satisfaction of the Initialisation or Action Simulation rules. It illustrates a process of problem detection and iterative development which is common at the meso and micro levels and which is facilitated by the simulation rules.

Stuttering Convergence. Since the number of atoms is finite, the stuttering action *Recruit* can only happen a finite number of times and will be disabled when $recruiters = placed$. This can be proved with the variant $W2 \mathrel{\widehat{=}} \#placed - \#recruiters$.

The above system specification *System*3 classifies atoms as being placed (and able to recruit atoms) or not being placed (and being able to move to a target position). The specification, however, does not detail the mechanism facilitating the recruiting and the movement.

Next, the development pushes the specification closer to the micro level. At this abstraction level, we model that atoms can only communicate with their immediate neighbours and can only move a short distance at each step. The design intuition of specification *System*4 is to refine the recruiting process (*Recruit*) and the moving of atoms (*Move*) in *System*3 by employing the virtual gradient mechanism described in Section 2. That is, atoms establish a virtual gradient of values (each value representing the distance from an atom requiring neighbours). An atom seeking a target position then moves in the direction indicated by a neighbouring atom with a lesser gradient value. For instance, in Figure 5, the atom with gradient value 3 (a position with an atom is denoted by a solid box) should move into one of the candidate positions represented by a dashed box. Each candidate position neighbours the atom with gradient value 2 (a neighbour of that with gradient value 3 with a lesser gradient value) and the atom with gradient value 1 (which is closer again to the seed atom). In this way, the atom with gradient value 3 can move progressively closer to the seed atom.

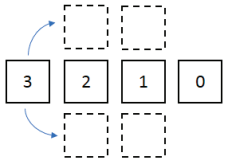


Figure 5: Atom movement according to a gradient.

The class *System*4 extends *System*3 with an additional variable $grad : Position \pfun \mathbb{N}$ mapping a subset of atoms to gradient values. Initially, no atoms have gradient values. A seed atom which is already placed and needs to recruit sets its gradient value to 0 and propagates the gradient to its neighbours. In order to ensure that gradients can propagate to all atoms, *System*4 has an invariant that all atoms are connected, i.e., $\forall\, p, q : atoms \bullet (p, q) \in nb^*$.

The retrieve relation $R3$ between $System3$ and $System4$ relates the abstract recruiting mechanism with the concrete gradient mechanism, i.e., an atom with gradient 0 in $System4$ is mapped to an atom in set *recruiters* in $System3$ which has vacant neighbouring target positions. The other variables remains the same in both systems.

$$R3 \mathrel{\widehat=} target_3 = target_4 \land placed_3 = placed_4 \land nb_3 = nb_4 \land (\forall p : Position \bullet p \in \operatorname{dom} grad_4 \land grad_4(p) = 0 \Rightarrow p \in recruiters_3 \land (\exists q : target_3 \setminus placed_3 \bullet (p, q) \in nb_3))$$

Following the retrieve relation, the abstract action *Recruit* can be simulated by a concrete change action *CreateGrad* in which a placed atom requiring neighbours creates a gradient by setting its gradient value to 0. The second to last line of the predicate in *CreateGrad* ensures that the action can only occur when the atom's gradient value is not already 0. This prevents an atom creating a gradient more than once (satisfying the abstract specification where *Recruit* can only occur once per atom).

The propagation of the gradients is specified by stuttering action *Propagate* which does not change the abstract state according to the retrieve relation. *Propagate* sets the gradient value of an atom (at position q) to 1 greater than the gradient value of its neighbour (at position p). If the atom at position q already has a gradient value then the action will be enabled only when its gradient value will be decreased. This ensures that the gradient value represents the shortest distance to an atom seeking a neighbour.

System4

$[Position]$

$target : \mathbb{F}\, Position$
$nb : Position \leftrightarrow Position$

$\forall p, q : target \bullet (p, q) \in nb^*$

$atoms : \mathbb{F}\, Position$
$placed : \mathbb{F}\, Position$
$grad : Position \pfun \mathbb{N}$

$\#atoms = \#target$
$placed \subseteq atoms \cap target$
$\operatorname{dom} grad \subseteq atoms$

Init

$\forall p, q : atoms \bullet (p, q) \in nb^*$
$placed \neq \varnothing$
$grad = \varnothing$

$\vdots$

CreateGrad

$\Delta(grad)$
$p, q : Position$

$(p, q) \in nb$
$p \in placed \land p \notin \operatorname{dom} grad$
$q \in target \setminus placed$
$grad' = grad \oplus \{p \mapsto 0\}$

Propagate

$\Delta(grad)$
$p, q : Position$

$p \in \operatorname{dom} grad$
$q \in atoms \land (p, q) \in nb$
$q \in \operatorname{dom} grad \Rightarrow grad(q) > grad(p) + 1$
$grad' = grad \oplus \{q \mapsto grad(p) + 1\}$

The abstract action *Move* is simulated by a concrete change action *Join* which specifies the final movement of an available atom to a target position. *Join* moves an atom (at position p) from one neighbouring position of a placed atom seeking a neighbour (at position q) to another neighbouring position of that atom which is part of the target (at

position r). The moving atom becomes placed. The third to last line of the predicate ensures that the action does not lead to the mass of atoms becoming disconnected; the actual mechanism by which this would be achieved is left to the micro level.

Join

$\Delta(atoms, placed)$
$p, q, r : Position$

$(p, q) \in nb \land (q, r) \in nb$
$grad(p) = 1 \land grad(q) = 0$
$p \in atoms \setminus placed \land r \in target \setminus placed$
$\forall a, b : atoms' \bullet (a, b) \in nb^*$
$atoms' = (atoms \setminus \{p\}) \cup \{r\}$
$grad' = \{p\} \lhd grad$
$placed' = placed \cup \{r\}$

Follow

$\Delta(atoms, grad)$
$p, q, r, s : Position$

$p \in atoms \setminus placed \land \{p, q, r\} \subseteq \mathrm{dom}\, grad$
$(p, q) \in nb \land (q, r) \in nb$
$s \notin atoms \land (q, s) \in nb \land (r, s) \in nb$
$grad(p) = grad(q) + 1$
$grad(q) = grad(r) + 1$
$\forall a, b : atoms' \bullet (a, b) \in nb^*$
$atoms' = (atoms \setminus \{p\}) \cup \{s\}$
$grad' = (\{p\} \lhd grad) \oplus \{s \mapsto grad(q)\}$

Follow moves an unplaced atom (at position p) from a neighbouring position of one atom with a smaller gradient value (at position q) to that of one of its neighbours (at position r) with an even smaller gradient value. The moving atom's gradient value (at its new position s) is then set to be equal to the atom's gradient value at position q. An atom at position p is allowed to move if it satisfies *Follow*'s guard (all but the last 2 lines of the predicate) which ensures:

(1) p is not placed and p, q and r have gradient values.

(2) q is a neighbour of p, r is a neighbour of q and s is a vacant position neighbouring both q and r.

(3) The gradient values of p, q and r form a decreasing sequence of values. This guarantees that the atom (at position p) moves towards a recruiter.

(4) The movement of atom from position p to s does not lead to disconnection of the atoms.

Finally, as mentioned in Section 2.2, we need to cancel gradients when they are no longer required. When a recruiter has all its neighbouring target positions filled, it should remove its gradient value. Additionally, atoms with a non-zero gradient value which no longer have a neighbour with a gradient value 1 less than theirs should remove their gradient value. The process is captured by action *DissipateGrad*.

DissipateGrad

$\Delta(grad)$
$p : Position$

$p \in \mathrm{dom}\, grad$
$grad(p) = 0 \Rightarrow (\nexists q : target \setminus placed \bullet (p, q) \in nb)$
$grad(p) \neq 0 \Rightarrow (\nexists q : \mathrm{dom}\, grad \bullet (p, q) \in nb \land grad(q) = grad(p) - 1)$
$grad' = \{p\} \lhd grad$

To ensure the correctness of the design, we need to check the action refinement simulation rules and fix any problems detected. According to the retrieve relation $R3$, all occurrences of *Recruit* and *Move* are change transitions in *System*3. All occurrences of *CreateGrad* and *Join* are change transitions in *System*4, and all occurrences of *Propagate*, *Follow* and *DissipateGrad* are stuttering transitions. Let Σ_n denote the state schema of class $System_n$, and let $System_n.Actions$ denote the disjunction of all actions of $System_n$.

Initialisation. The condition is true since the initial state schema of *System*4 is related to that of *System*3 under retrieve relation $R3$, i.e., $\forall \Sigma_4 \bullet (\forall p, q : atoms\_4 \bullet (p, q) \in nb_4^*) \land placed_4 \neq \varnothing \land grad_4 = \varnothing \Rightarrow \exists \Sigma_3 \bullet placed_3 \neq \varnothing \land recruiter_3 = \varnothing \land R3$.

Action Simulation. The guard of *CreateGrad* (all lines but the last) implies the guard of *Recruit* according to $R3$. Also, the final line $grad_4' = grad_4 \oplus \{p \mapsto 0\}$ of *CreateGrad* corresponds to the line $recruiters_3' = recruiters_3 \cup \{p\}$ of *Recruit*. It is trivial to conclude that *CreateGrad* simulates *Recruit*. The proof for *Join* simulating *Move* follows similarly.

It is also trivial to prove that stuttering actions *Propagate*, *Follow* and *DissipateGrad* refine $ID_{System3}$ since their post-states are related to the exact abstract states related to their pre-states under $R3$.

Termination. This condition is satisfied when

$\forall \Sigma_3, \Sigma_4 \bullet R3 \Rightarrow (\neg \operatorname{pre} System4.Actions \Rightarrow \neg \operatorname{pre} System3.Actions)$.

After the modification to the initial schema of *System*3, we have $\neg \operatorname{pre} System3.Actions = (target_3 = placed_3)$. After trivial deduction, the conclusion we need to establish is $\forall \Sigma_4 \bullet (\neg \operatorname{pre} System4.Actions) \Rightarrow (target_4 = placed_4)$. This can be proved by checking the guard of each action in *System*4. Doing so, we find a problem when we attempt to prove $\neg \operatorname{pre} Follow \Rightarrow (target_4 = placed_4)$. This problem occurs when all atoms at a position satisfying the constraints of r in *Follow* have no vacant neighbouring positions. The problem is discussed by Støy [14] who suggests assuming the target configuration is in the form of a 'scaffold' with vacant positions around any positions in which atoms are required. We adopt a similar approach by adding the following invariants to System4.

$\forall p, q : atoms \bullet (p, q) \in nb \Rightarrow (\exists r : Position \setminus atoms \bullet (p, r) \in nb \land (q, r) \in nb)$

$\forall p, q : target \bullet (p, q) \in nb \Rightarrow (\exists r : Position \setminus target \bullet (p, r) \in nb \land (q, r) \in nb)$

With this invariant, the guard of *Follow* is always *true* until $target_4 = placed_4$, which satisfies the termination rules. The modification does not affect the proofs of Initialisation or Action Simulation.

Stuttering Convergence. According to the retrieve relation, the stuttering actions in *System*4 are *Propagate*, *Follow* and *DissipateGrad*. To prove stuttering convergence, we need to show that these actions cannot be executed infinitely. Let $Stutt_4 = Propagate \lor Follow \lor DissipateGrad$, we need to prove:

$\forall \Sigma_4, \Sigma_4' \bullet \exists N : \mathbb{N} \bullet Stutt_4^N \Rightarrow \neg(\operatorname{pre}\ Stutt_4)'$

In the following discussion, we find an upper bound for N by showing that the number of executions of each stuttering action, when interleaved with the other stuttering actions, has an upper bound.[1]

[1] It is enough to prove the existence of an upper bound. For simplicity, we assume the worst case in every step of execution. Hence the actual upper bound of N is smaller than the number we obtain.

Let n be the number of atoms and m be the number of recruiters with gradient 0 in the pre-state. We use the notation $\overline{N_P}$, $\overline{N_F}$ and $\overline{N_D}$ to denote the upper bounds on the numbers of executions of *Propagate*, *Follow* and *DissipateGrad* respectively.

The execution of *DissipateGrad* decreases a variant W_D which is a set containing

(1) any recruiter with gradient 0 which no longer has the requirement of recruiting (all its neighbouring target positions have been filled).

(2) any atom with gradient greater than 0 which does not have a neighbour with a gradient value 1 less than its own.

$$W_D \mathrel{\widehat{=}} \{p \mid grad(p) = 0 \wedge \nexists r : target \setminus placed \bullet (p, r) \in nb\}$$
$$\cup \{p \mid grad(p) \neq 0 \wedge \nexists q : \mathrm{dom}\, grad \bullet (p, q) \in nb \wedge grad(q) = grad(p) - 1\}$$

Note that when a recruiter dissipates its gradient value, it implies that all its neighbouring target positions have been filled, hence it will never recruit again. Therefore, the maximum number of executions of *DissipateGrad* for cancelling a recruiter's gradient value is n (assuming every atom becomes a recruiter). After a recruiter's gradient value is cancelled, the atoms whose gradient values form the gradient to this recruiter also dissipate their values through executing *DissipateGrad*. The maximum number of executions of *DissipateGrad* to achieve this is $n - 1$ (assuming every other atom was involved in forming the gradient). In summary, the maximum number of executions of *DissipateGrad* is $\overline{N_D} = n(n - 1)$. Note that executions of *Propagate* do not increase W_D since they do not increase the number of atoms having gradient 0 or not having a neighbour with a gradient 1 less than its own. Similarly, executions of *Follow* do not increase W_D either.

The execution of *Propagate* decreases a variant W_P defined as follows.

$$W_P \mathrel{\widehat{=}} n * \#\{p \mid p \in atoms \wedge p \notin \mathrm{dom}\, grad\} + \sum_{q \in dom\ grad} grad(q)$$

The variant W_P has a lower bound when every atom has a gradient leading to its nearest recruiter. For every atom that is not a recruiter, the number of executions of *Propogate* to update its gradient value is at most m (in the worst case where the gradient from the nearest recruiter arrives last). Hence, without considering the other stuttering actions, *Propagate* can be executed for at most $m(n - m)$ times. The executions of *Follow* cannot increase W_P since they reduce the gradient of the moving atom. The execution of *DissipateGrad* can increase W_P whenever it removes the gradient value of an atom. For every atom which dissipates its gradient value, *Propagate* will be executed at most $m-1$ times to reset its gradient value to lead to another recruiter. The gradient value reset by such a *Propagate* execution will not be removed until the recruiter is cancelled. Hence, the upper bound of additional execution number of *Propagate* equals the upper bound of the execution number of *DissipateGrad* (denoted by $\overline{N_D}$) times $(m-1)$. In summary, the upper bound on executions of *Propagate* is captured by $\overline{N_P} = m(n - m) + \overline{N_D}(m - 1)$.

The execution of *Follow* decreases the following variant W_F.

$$W_F \mathrel{\widehat{=}} \sum_{p \in P} d(p, q) \text{ provided } grad(q) = 0 \wedge grad(p) = d(p, q)$$
$$+ n * \#\{p \mid p \in atoms \wedge p \notin \mathrm{dom}\, grad\}$$

where $P \mathrel{\widehat{=}} \{p \mid p \in atoms \setminus placed \wedge p \in \mathrm{dom}\, grad\}$ and $d(p, q) \geq 0$ is the distance between position p and q, i.e., $d(p, q) = i$ iff i is the minimum number such that

$(p, q) \in nb^i$.

The execution of *Follow* moves an unplaced atom closer to the recruiter that its gradient leads to until it is a neighbour of the recruiter (at this moment its gradient equals 1). Therefore, the variant W_F has a lower bound when every unplaced atom has reached the neighbourhood of its recruiter enabling the change action *Join*. For any unplaced atom i, the worst case is when the distance between itself and its recruiter is $n-1$ (the maximum distance given that the atoms are connected). Hence, the upper bound on executions of *Follow* without any other actions is $(n-m)(n-2)$ (the gradient decreases from $n-1$ to 1 for $n-m$ unplaced atoms). The execution of *Propagate* cannot increase W_F since it only updates a gradient value of an unplaced atom when there is a nearer recruiter. The execution of *DissipateGrad* can increase W_F when it removes the gradient of an unplaced atom whose recruiter's gradient is cancelled. In this case, a new gradient will be assigned to the unplaced atom by *Propagate* indicating a new recruiter. The action *Follow* can be executed for at most $n-2$ times to move the unplaced atom to its new recruiter. In the worst case, this process can be repeated $m-1$ times. Hence the additional number of executions of *Follow* is at most $(m-1)(n-2)$. In summary, we have $\overline{N_F} = (n-1)(n-2)$.

Based on the above analysis, we have an upper bound of N defined by $\overline{N} = \overline{N_F} + \overline{N_D} + \overline{N_P}$, which means the stuttering behaviour of *System*4 converges.

4.3. Micro level

The design intuition of the micro-level specification is to introduce the local state of every atom into the model and simulate every action of the meso-level specification with a single atom's local action or an interaction between an atom and the environment. At this level, global information cannot be accessed directly by a single atom. Information is instead exchanged via communications between atoms. In our case study we have exactly one type of agent, an atom.

Following the framework proposed in Section 3.3, we first define a class *Atom*. The state of this class is given below. *Position* and *nb* play the same role as in the meso-level specifications. $\mathbb{N}_\infty$ is a type to denote the gradient value of an atom; it is either a natural number or the symbol ∞ denoting that no numerical gradient value has been established. A relation $<$ is defined to relate gradient values: $n_1 < n_2$ iff $n_2 = \infty$ and n_1 is a number, or both n_1 and n_2 are numbers and n_1 is less than n_2. This relation is used both in establishing gradient values, and in determining whether an atom can move, i.e., whether the connectedness of atoms will be maintained by the move.

The type **message** of *Atom* defines the types of messages used in the system. For example, *request* is the type of message which is sent by an atom intending to move and *response* is the type of messages which is sent by an atom responding to a *request* message. The parameters in $\langle\langle\rangle\rangle$ are the types of values carried by the messages. We will explain each message type in detail when we get to the specifications of the actions.

Following Støy [14], we assume that the atoms that are placed in a target position have a copy of the target (*target*), and know their position within it (*pos*). All other atoms will have an empty target, i.e., *target* $= \varnothing$, in which case the value of *pos* is meaningless. As in the meso-level specification, the target is in the form of a 'scaffold'. Placed atoms also know which of their neighbouring target positions are filled (*filled*).

At this level, each atom is associated with a gradient value in terms of a local variable *grad*. The usage of the gradient value is the same as in the meso-level specification; the difference is that it is maintained by the atom. In the absence of the global view, each atom now has a local variable *next* recording its neighbouring atoms which have a smaller gradient value. It is used for indicating the direction in which the atom should move. Each atom also has a variable *dependent* recording its neighbouring atoms which have a larger gradient value. It is used for ensuring that the movement of atoms does not violate the connection of the atoms. A boolean state variable *joined* indicates that an atom has joined the target position and a boolean state variable *notified* indicates that it has notified its neighbours to update their filled set.

Atom

$[Position]$

$nb : Position \leftrightarrow Position$

$\mathbb{N}_\infty ::= num\langle\langle \mathbb{N} \rangle\rangle \mid \infty$

$\_ < \_ : \mathbb{N}_\infty \leftrightarrow \mathbb{N}_\infty$

$$\forall n_1, n_2 : \mathbb{N}_\infty \bullet n_1 < n_2 \Leftrightarrow (n_2 = \infty \land n_1 \neq \infty \lor n_1 \neq \infty \land n_2 \neq \infty \land num^\sim(n_1) < num^\sim(n_2))$$

$$\textbf{message} ::= grad\langle\langle \mathbb{N}_\infty \rangle\rangle \mid request \mid response\langle\langle Atom \times \mathbb{N}_\infty \rangle\rangle \mid newdep\langle\langle Atom \rangle\rangle \mid moving \mid dissipate \mid target\langle\langle \mathbb{F}\, Position \times Position \rangle\rangle \mid joined\langle\langle Position \rangle\rangle \mid ack\langle\langle Position \rangle\rangle$$

$pos : Position$
$target, filled : \mathbb{F}\, Position$
$grad : \mathbb{N}_\infty$
$next, dependent : \mathbb{F}\, Atom$
$joined, notified : \mathbb{B}$

$$target \neq \varnothing \Rightarrow pos \in target$$
$$\forall p, q : target \bullet (p, q) \in nb \Rightarrow (\exists r : Position \setminus target \bullet (p, r) \in nb \land (q, r) \in nb)$$

Init

$$filled = \varnothing \land grad = \infty \land next = dependent = \varnothing$$
$$(joined \Leftrightarrow target \neq \varnothing) \land \neg notified$$

$\vdots$

The use of the agent specification *Atom* as a type (in the declaration of *next* and *dependent*) is borrowed from Object-Z where classes are similarly used as types. As in Object-Z, instances of agent specifications in MAZE are *references* to the agents. Such a reference is independent of the agent's state and does not change as the agent performs actions. In an implementation of our case study, we would not expect an atom to store such references. Their use is simply an abstraction for another means of referring to particular neighbouring atoms, such as the port through which they communicate.

Initially, *filled* is empty, there is no gradient value and no atoms in *next* and *dependent*, *notified* is false and *joined* is true only for atoms which have a copy of the target. The action schemas of the class *Atom* will be defined in the next subsection.

The system specification at the micro level is captured by the following class *System*5.

*System*5

$$atoms : \mathbb{T}\, Atom$$
$$position : Atom \rightarrowtail\!\!\!\!\!\!\rightarrow Atom.Position$$

$$\text{dom}\, position = \text{dom}\, atoms$$
$$\forall\, a, b : \text{dom}\, atoms \bullet$$
$$\quad a.nb = b.nb \land (a, b) \in atoms \Leftrightarrow (position(a), position(b)) \in a.nb^*$$
$$\forall\, a : \text{dom}\, atoms \bullet a.target \neq \varnothing \Rightarrow position(a) = a.pos$$
$$\forall\, p, q : \text{ran}\, position \bullet$$
$$\quad (p, q) \in a.nb \Rightarrow (\exists\, r : Position \setminus \text{ran}\, positions \bullet (p, r) \in a.nb \land (q, r) \in a.nb)$$

Init

$$\forall\, a, b : atoms \bullet (position(a), position(b)) \in a.nb^*$$
$$\exists_1\, a : \text{dom}\, atoms \bullet a.joined$$

⋮

A topology $atoms : \mathbb{T}\, Atom$ is defined over the atoms to establish the spatial properties and the communications links of the system. An injective function *position* mapping agents to their positions is defined as part of the system state. Note that an atom cannot directly access the system variable *position*. In order to capture the fact that only neighbouring atoms can communicate with each other, we include an invariant relating *position* to the topology *atoms* so that atoms only receive messages sent from their neighbouring atoms. The third invariant indicates that when an atom is placed (i.e., its *target* variable is non-empty) its position corresponds to the position (*pos*) it stores as part of its state. The last invariant states a scaffold assumption like that placed on *atoms* in *System*4.[2]

Initially, all atoms are connected and, following Stoy and Nagpal [14, 15], there is exactly one atom in the target.

To develop the micro-level specification from the meso-level specification, we use the following retrieve relation between the states of *System*4 and *System*5. It reflects our development intuition regarding which observations are considered as consistent in both systems.

$$\begin{aligned} R4 \mathrel{\widehat=} & (\forall\, a : \text{dom}\, atoms_5 \bullet a.target \neq \varnothing \Rightarrow a.target = target_4) \land a.nb = nb_4 \land \\ & atoms_4 = \{a : \text{dom}\, atoms_5 \bullet position_5(a)\} \land \\ & placed_4 = \{a : \text{dom}\, atoms_5 \mid a.joined \bullet position_5(a)\} \land \\ & grad_4 = \{a : \text{dom}\, atoms_5 \mid a.grad \neq \infty \bullet (position_5(a), a.grad)\} \end{aligned}$$

[2]The variable *position* is required to relate the local constant *nb* and the local variable *pos* of *Atom* to the topology described by the variable *atoms*. Future work will look at extending MAZE actions to specify movement in 2D or 3D space directly, without the need for such local variables.

According to $R4$, the local *target* variable of every placed atom is consistent with $target_4$. The positions of atoms specified by $atom_4$ in *System*4 is captured by the range of $position_5$ in *System*5. $placed_4$ is captured by the positions of atoms with *joined* true. The gradient function $grad_4$ is captured by the positions and gradients of all atoms a with $a.grad \neq \infty$.

Since $R4$ relates each of the abstract variables to the concrete state, they are observable and hence each abstract action (since it changes at least one of these abstract variables) is a change action.

Initialisation. The Initialisation condition holds since initially in both *System*4 and *System*5 the mass of atoms is connected, no atom has a gradient value, and there is at least one atom placed in a target position (exactly one in *System*5).

The micro-level agent actions must simulate the actions of *System*4. Hence we verify the Action Simulation rule during the design of the actions in this section. Below we focus on agent actions for creating and propagating gradients, and for moving. Actions for dissipating gradients and joining the target can be defined in a similar fashion (see the Appendix).

4.3.1. Gradients

In the meso-level specification, we have actions *CreateGrad* and *Propagate* to generate and propagate gradients. At the micro level, an agent action *NewGrad* is designed to simulate *CreateGrad*. It allows an agent which has joined the target and has unfilled neighbouring positions to set its gradient value to zero and broadcast a *grad* message. The progress predicate ensures that its previously sent *joined* message has been received by each of the atom's neighbours, and the atom has received their acknowledgments.[3]

Similarly, the meso-level action *Propagate* is simulated by the following agent action *SetGrad* which models an agent, on receiving a *grad* message with a gradient value g less than its own, setting its own value to g and broadcasting a *grad* message with value $g + 1$.

NewGrad

$\Delta(grad)$

$\mathbf{progress}(\{joined(pos)\},$
$\quad \{p : Position \bullet ack(p)\})$
$notified$
$filled \neq \{p : target \mid (pos, p) \in nb\}$
$grad = \infty \land grad' = 0$
$\mathbf{send}(grad(1))$

SetGrad

$\Delta(grad, next)$
$g : \mathbb{N}_\infty$
$a : Atom$

$\mathbf{receive}(grad(g), a)$
$g < grad$
$grad' = g$
$g = grad - 1 \Rightarrow next' = next \cup \{a\}$
$g < grad - 1 \Rightarrow next' = \{a\}$
$\mathbf{send}(grad(g + 1))$

Action Simulation. The proof that the implicit *System*5 action $[]\ a : \mathrm{dom}\, atoms \bullet a.NewGrad$ simulates *CreateGrad* can be established by checking their guards and effects. *CreateGrad* is enabled when a placed atom with a non-zero gradient value has an empty target position in its neighbourhood, i.e.,

[3] More details about variable *notified*, *joined* and *ack* messages and their affects on *filled* are provided by agent actions *Fill* and *UpdateFilled* in the Appendix.

$\exists\, p, q : \mathit{Position} \bullet (p, q) \in \mathit{nb}_4 \land p \in \mathit{placed}_4 \land p \notin \operatorname{dom} \mathit{grad}_4 \land q \in \mathit{target}_4 \setminus \mathit{placed}_4$

Letting $a.\mathbf{progress}$ denote the progress predicate (as defined in Definition 5) for atom a, the guard of $a.\mathit{NewGrad}$ is

$a.\mathbf{progress} \land a.\mathit{notified} \land a.\mathit{filled} \neq \{p : \mathit{target} \mid (\mathit{pos}, p) \in \mathit{nb}\} \land a.\mathit{grad} = \infty$

Let $p = \mathit{position}_5(a)$. It is trivial to deduce that $a.\mathit{notified} \Rightarrow a.\mathit{joined}$ (from the definition of action *Fill* in the Appendix). According to $R4$, $a.\mathit{joined} \land a.\mathit{grad} = \infty \Rightarrow p \in \mathit{placed}_4 \land p \notin \operatorname{dom} \mathit{grad}_4$. The **progress** predicate ensures that the atom a has received all acknowledgments to its *joined* messages from its neighbours and updated its *filled* variable to have a complete view of which of its neighbouring target positions are filled. Hence we have $a.\mathbf{progress} \land a.\mathit{filled} \neq \{p : \mathit{target} \mid (\mathit{pos}, p) \in \mathit{nb}\} \Rightarrow \exists\, q : \mathit{Position} \bullet (p, q) \in \mathit{nb}_4 \land q \in \mathit{target}_4 \setminus \mathit{placed}_4$. Therefore, $a.\mathit{NewGrad}$ has a stronger guard than *CreateGrad*, i.e., $\operatorname{pre}\ (\,[\!]\ a : \operatorname{dom} \mathit{atoms} \bullet a.\mathit{NewGrad}) \Rightarrow \operatorname{pre} \mathit{CreateGrad}$. The effects of both actions result in an atom's gradient value being set to 0, i.e., $a.\mathit{grad}' = 0 \Rightarrow \mathit{grad}'_4 = \mathit{grad}_4 \oplus \{p \mapsto 0\}$. Therefore we have $[\!]\ a : \operatorname{dom} \mathit{atoms} \bullet a.\mathit{NewGrad}$ simulates *CreateGrad*. For simplicity, in the rest of the paper we use the notation $\mathit{atoms}.\mathit{Action}$ to denote the system action $[\!]\ a : \operatorname{dom} \mathit{atoms} \bullet a.\mathit{Action}$.

The proof that $\mathit{atoms}.\mathit{SetGrad}$ simulates *Propagate* can be conducted as follows. The effects of both actions set an atom's gradient value to one more than that of a neighbouring atom. We only need to prove that $R4 \land \operatorname{pre} \mathit{atoms}.\mathit{SetGrad} \Rightarrow \operatorname{pre} \mathit{Propagate}$. *Propagate* is enabled when an atom at position p has a neighbouring atom at position q with a gradient value at least 2 more than its own, i.e.,

$p \in \operatorname{dom} \mathit{grad}_4 \land q \in \mathit{atoms}_4 \land (p, q) \in \mathit{nb}_4 \land (q \in \operatorname{dom} \mathit{grad}_4 \Rightarrow \mathit{grad}_4(q) > \mathit{grad}_4(p) + 1)$

$\mathit{atoms}.\mathit{SetGrad}$ is enabled whenever a message $(\mathit{grad}(g), b, a)$, for some $b : \operatorname{dom} \mathit{atoms}$, is in the **buffer** with g less than $a.\mathit{grad}$. That is, letting $a.\mathbf{receive}(m, b)$ denote the predicate for atom a receiving message m,

$\exists\, b : \operatorname{dom} \mathit{atoms} \bullet a.\mathbf{receive}(\mathit{grad}(g), b) \land g < a.\mathit{grad}$

Since *grad* messages are only sent by *NewGrad* and *SetGrad* and these actions send values 1 greater than the *grad* value of their sender, *SetGrad* only occurs when the sender's gradient value is at least 2 less than its own. Hence, we have

$$R4 \land (\exists\, a, b : \operatorname{dom} \mathit{atoms} \bullet a.\mathbf{receive}(\mathit{grad}(g), b) \land g < a.\mathit{grad}) \Rightarrow$$
$$\exists\, p, q : \mathit{Position} \bullet$$
$$p \in \operatorname{dom} \mathit{grad}_4 \land q \in \mathit{atoms}_4 \land (q \in \operatorname{dom} \mathit{grad}_4 \Rightarrow \mathit{grad}_4(q) > \mathit{grad}_4(p) + 1)\,.$$

To prove the simulation, we still need to ensure that receiving such a *grad* message implies b is a neighbour of a, i.e.,

$a.\mathbf{receive}(\mathit{grad}(g), b) \Rightarrow (\mathit{position}(a), \mathit{position}(b)) \in \mathit{nb}_4$ or equivalently,

$\forall\, a, b : \operatorname{dom}\ \mathit{atoms}, g : \mathbb{N}_\infty \bullet (\mathit{grad}(g), b, a) \sqsubseteq \mathbf{buffer} \Rightarrow b \in \mathit{atoms}(a)$

This requires that (i) a $\mathit{grad}(g)$ message can be sent from b to a only when b is a neighbour of a with gradient value g, and (ii) after sending such a message b stays in the neighbourhood of a until the message is received. The former is ensured in *System*5 by the restriction on the agent topology that atoms can only send messages to their neighbours (see the state schema of *System*5 in Section 4.3). The latter requires any action which moves an agent which may have sent a *grad* message to have a **progress** statement

stating that all such messages have been received. This requirement is considered in the next subsection.

4.3.2. Moving

The meso-level action *Follow* requires an atom (e.g., a) to move from being a neighbour of one atom b to being a neighbour of another c which is closer to a recruiter. More specifically, b is a neighbour with a lower gradient value than a and c is a neighbour of b having a lower gradient value. At the micro level, this action is simulated by the following process. First, an atom a uses local variable *next* to record neighbours with a lower gradient values when it performs *SetGrad* (see its definition in Section 4.3.1). Second, the neighbouring atom b is chosen from $a.next$ and the atom c is chosen from $b.next$. As $b.next$ is not a part of the local state of the atom a, a communication between a and b is required to guide the movement. Hence we introduce two communication actions *Request* and *Respond*. *Request* enables an atom to request the reference to an atom in the *next* set of one of its neighbours. *Respond* models the response to such a request. (The guard of *Request* and the message *newdep* sent by *Response* are described later in this subsection.)

Request

$b : Atom$

$\mathbf{progress}(\{grad(grad), request\}, \{grad(grad + 1), response\})$
$target = \varnothing \land dependent = \varnothing$
$b \in next$
$\mathbf{send}(request, b)$

Respond

$a, c : Atom$

$\mathbf{receive}(request, a)$
$c \in next$
$\mathbf{send}(response(c, grad), a)$
$\mathbf{send}(newdep(a), c)$

When a response is received, the requesting atom can move. It sets its gradient value to the received gradient g and updates its *next* to include just the atom to whose neighbourhood it moves (atom c). It also broadcasts to its neighbours that it is moving. This is captured by agent action *Move*. When the agent action *Move* is performed, a synchronised system action $Move < a : \mathrm{dom}\, atoms \bullet a.Move >$ changes the topology: an agent a moves to a position s neighbouring atoms b and c. The actual values of b and c are constrained by those of $a.Move$ whose common-named declarations are equated with those of *Move*.

Move

$\Delta(grad, next)$
$g : \mathbb{N}_\infty$
$b, c : Atom$

$\mathbf{receive}(response(c, g), b)$
$grad' = g \land next' = \{c\}$
$\mathbf{send}(moving)$

$Move < a : \mathrm{dom}\, atoms \bullet a.Move >$

$\Delta(position)$
$b, c : \mathrm{dom}\, atoms$
$s : Position$

$s \notin \mathrm{ran}\, position$
$(position(b), s) \in b.nb$
$(position(c), s) \in c.nb$
$position' = position \oplus \{a \mapsto s\}$

Action Simulation. The meso-level action *Follow* is intended to be simulated by the micro-level system action *Move*. For simplicity, in the following proof, we refer to atom

a (at position p) as the moving atom, atom b (at position q) as the atom which responds to a's request, and atom c (at position r) as the atom that b includes in its response.

From the definition of *Move* we can conclude that its guard is atom a receiving a response message from b and that there is a vacant position s neighbouring atoms b and c. It is trivial to show that the constraints regarding position s at the micro level imply the meso-level constraints $s \notin atoms_4 \wedge (q, s) \in nb_4 \wedge (r, s) \in nb_4$. To ensure that **receive**$(response(c, g), b)$ implies the rest of the constraints in the guard of *Follow* requires that:

(a) the guards of $a.Request$ and $b.Response$ imply that $a.grad = b.grad + 1$ and $b.grad = c.grad + 1$;

(b) the movement of atom a does not break the connection of the atoms.

The guard of action $a.Request$ includes $target = \varnothing$ which implies that it is not placed and $b \in a.next$ which implies that $a.grad = b.grad + 1$ (according to the definition of action *SetGrad*). Similarly, $c \in b.next$ implies that $b.grad = c.grad + 1$.

The **progress** condition in $a.Request$ indicates that all *grad* messages from and to it have been received, which implies that atom a knows all the gradient values of its neighbours before it can move (this constraint completes the proof that *SetGrad* simulates *Propagate* in the previous subsection). The **progress** condition also ensures that atom a does not re-send *request* before all *request* messages from a and *response* messages to a have been handled (this constraint is useful when we consider the Stuttering Convergence rule). The implementation of such a **progress** constraint can be the atom waiting for a worst-case response time dependent on the communication medium. The progress condition enables us to abstract from such an implementation-dependent timing constraint.

In order to ensure condition (b), we adopt a solution which allows only the atom with the greatest gradient among its neighbours to move (and hence prohibits an atom moving if another atom depends on it to remain connected to the mass; see Figure 3). This solution is realised by only letting an atom a move when $a.dependent = \varnothing$. In general, an atom b will be added to $a.dependent$ in the following situations: (i) b has a greater gradient than a, or (ii) b intends to move to the neighbourhood of a. The latter is signalled by the *newdep* message sent as part of *Respond*. The following two actions are introduced to capture the two situations.

*AddDependent*1

$\Delta(dependent)$
$g : \mathbb{N}_\infty$
$b : Atom$

---

receive$(grad(g), b)$
$g > grad$
$dependent' = dependent \cup \{b\}$

*AddDependent*2

$\Delta(dependent)$
$b, c : Atom$

---

receive$(newdep(b), c)$
$dependent' = dependent \cup \{b\}$

With the above solution, we can prove condition (b), i.e., if atoms a, b and c are connected, then any enabled movement cannot break the connection. Let $a.grad = b.grad + 1$, $b.grad = c.grad + 1$, $a.next = \{b\}$ and $b.next = \{c\}$. From *AddDependent*1, we have $b.dependent = \{a\}$ and $c.dependent = \{b\}$. So b and c cannot move. Atom

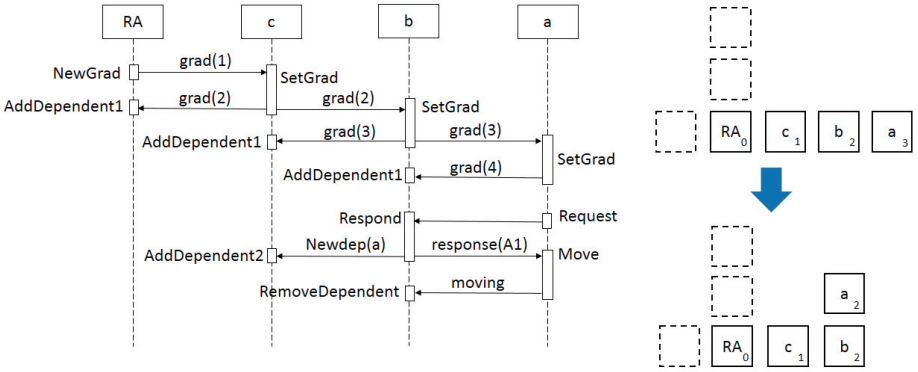


Figure 6: Sequence diagram for gradient setting process and moving process.

a can move towards c after $a.Request$ and $b.Response$ are performed. According to the definition of *Move*, the destination position of a's movement is a neighbour of both b and c. Hence, after this move, a, b and c are still connected.

Figure 6 shows the gradient setting phase and the movement of the atom a with the highest gradient towards its recruiter RA. The right hand side shows the changes of the system configuration where the number at bottom-right corner of an atom indicates its current gradient value, the dashed boxes represent empty target positions.

4.3.3. Checking Termination

The Termination rule requires us to prove that

$$\forall \Sigma_4, \Sigma_5 \bullet R4 \Rightarrow (\text{pre } System4.Actions \Rightarrow \text{pre } System5.Actions)$$

In Section 3.2, we have pre $System4.Actions = (target_4 \neq placed_4)$. In order to ensure the termination condition, we need to introduce additional stuttering actions which complement the gap between every simulating change actions in $System5$ so that no deadlocks can happen before the atoms reach the target shape. In this section we only discuss the stuttering actions associated with the gradient propagation process and the agent moving process and prove that they do not cause deadlocks. The other actions are considered in the Appendix.

Note that the guard of *Request* requires that all *grad* messages associated with the requesting atom have been handled (i.e., do not appear in the buffer). To ensure that *Request* can be enabled, we need to have actions to receive all *grad* messages. For any atom a, we have action $a.SetGrad$ and $a.AddDependent1$ to handle $grad(g)$ messages with $g < a.grad$ and $g > a.grad$, but we still lack an action to handle the *grad* messages with $g = a.grad$. Hence, we introduce the following action *IgnoreGrad* to model an atom receiving such a *grad* message and ignoring it.

IgnoreGrad	*RemoveDependent*
$g : \mathbb{N}_\infty$ $a : Atom$	$\Delta(dependent)$ $a : Atom$
receive$(grad(g), a)$ $grad = g$	**receive**$(moving, a)$ $dependent' = dependent \setminus \{a\}$

The guard of *Request* also requires that $dependent = \varnothing$. At this stage, we only have actions *AddDependent*1 and *AddDependent*2 to add dependents but lack an action to remove dependents. This will obviously lead to deadlock. A dependent a should be removed when it moves from the neighbourhood of an atom. Hence, on receiving a *moving* message from a (sent by the *Move* action), the receiving atom can remove a from its *dependent* set if necessary. This is captured by the action *RemoveDependent*.

Another kind of deadlock can happen at the micro level because of the interleaving of the agent actions. This case happens when an atom b sends a *response* message to an atom a, but before a receives the message, the destination position s has been occupied by another moving atom d. In this case, the system action *Move* is not enabled for a since there is not a position s satisfying

$$s \notin \operatorname{ran} position \land (position(b), s) \in b.nb \land (position(c), s) \in c.nb$$

Hence, the *response* message remains in the buffer blocking the atom a's consequent attempts to move (since the guard of $a.Request$ requires all *response* messages to a to be handled). To fix this, we add an additional agent action *CancelMove* (and an associated system action) to absorb the redundant *response* messages so that the atom can re-initiate its request to move.

CancelMove

$g : \mathbb{N}_\infty$
$b, c : Atom$

receive$(response(c, g), b)$

CancelMove $<a : atoms \bullet a.CancelMove>$

$b, c : \operatorname{dom} atoms$

$\nexists s : Position \bullet$
$s \notin \operatorname{ran} position \land (position(b), s) \in nb \land$
$(position(c), s) \in nb$

Figure 7 shows the case where the deadlock is possible and illustrates the solution using *CancelMove*. In the figure, atom d moves to the dashed position before atom a receives the response from atom b. Then atom a cannot perform *Move* since the destination position is blocked. Through performing *CancelMove*, atom a can continue the execution by sending another request to another atom f and moving to the dashed position at the bottom.

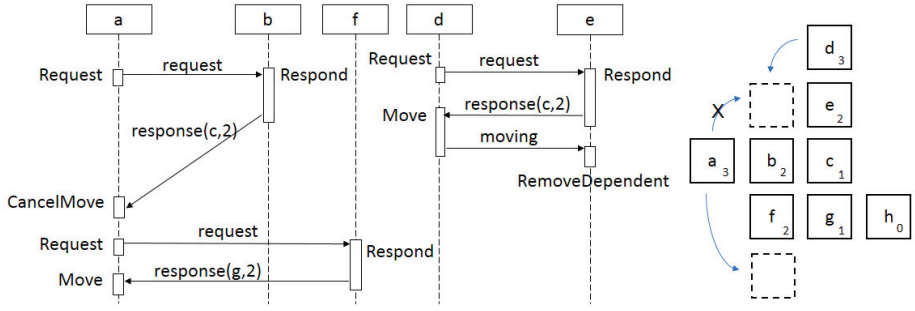


Figure 7: Sequence diagram for the deadlock case and its solution.

4.3.4. Checking Stuttering Convergence

To prove Stuttering Convergence rule at the micro level, we need to ensure that the stuttering actions of *System*5 cannot be executed an infinite number of times. After reviewing the stuttering actions defined in previous subsections and the Appendix, we find that most stuttering actions have a **receive** predicate in their guard. For example, the guard of action *b.Respond* includes $\exists\, a : \mathrm{dom}\, atoms \bullet \mathbf{receive}(request, a)$. We can hence define a variant

$$W_{b.Respond} \mathrel{\widehat{=}} \sum_{a \in dom\ atoms} \mathbf{buffer} \,\#\, (request, a, b)$$

where $\mathbf{buffer} \,\#\, m$ is the number of times m appears in the bag **buffer** [13].

It has a lower bound 0 when all requests in the buffer are handled. To ensure *b.Respond* can be only executed a finite number of times, we ensure that there can be only a finite number of *request* messages for b in the buffer. The *request* messages can be only sent by b's neighbours performing *Request*. Note that an atom has a finite number of neighbours and for each neighbour a the **progress** predicate of *a.Request* prevents it sending additional *request* messages to b before it receives b's response and performs the change action *a.Move*.

When a performs *CancelMove* since the destination position is blocked, according to the scaffold assumption and the fact that there are only a finite number of moving atoms (say m), the worst case of performing *b.Respond* is $m - 1$ times (when all other moving atoms happen to occupy the destination position a is referred to).[4] A similar analysis can be done for the action *a.CancelJoin* (see Appendix) which may cause additional executions of *a.Request*. The other stuttering actions cannot send *request* messages nor cause additional *a.Request* actions, so they have no influence on the variant. Therefore, for any atom b, the stuttering action *b.Respond* can be only executed a finite number of times.

The other stuttering actions with guards including **receive** predicates can be analysed in a similar way. They depend on the actions which send the corresponding messages. Checking all agent actions which can send a message, we find that these actions can only generate a finite number of messages. For example, the number of messages that action *NewGrad* generates equals the number of its current neighbours, which is finite. Furthermore, it can be only executed once. These *grad* messages sent to the buffer would be received by neighbouring atoms and trigger their *SetGrad* action. *SetGrad* can only be executed when it has received a *grad* message having a gradient value less than its own. Since it has a finite number of neighbours (say n), the gradient would be set at most n times before a movement of a neighbour atom, which is a change action, occurs.

The stuttering actions which do not have a **receive** statement are *Request* and *Fill*. As our analysis above shows, the upper bound of executing *Request* for an atom a is the number of moving atoms. The action *Fill* happens after an atom has joined the target position but has not sent out a *joined* message. From its definition (see the Appendix) we can trivially conclude that for any atom a, *a.Fill* can be executed only once since there is no way to change *a.notified* from *true* to *false*.

In summary, all stuttering actions in *System*5, when interleaved with other stuttering actions, can only be executed a finite number of times, which implies stuttering

[4] This also proves that for any atom a, the upper bound of executing *a.Request* is m.

convergence of the system.

5. Discussion and related work

The formal specification of agents and MAS has been explored by many researchers. Much of this work focusses on the high-level formalisation and verification of agents' "intelligent" behaviours. Hindriks et al. [24] presents an agent programming theory which provides both an agent programming language and a corresponding verification logic to verify such agent programs. Fisher [25] formalises the deliberation of agents using executable temporal logic. The deliberation behaviour is captured by modifying the execution rules handling the agents' temporal goals. The BDI model (of agents which are driven by their beliefs, desires and intentions) [2, 26] is a popular model for intelligent agents. A series of BDI logics are proposed for describing and reasoning about the mental attitudes of BDI agents [27, 28].

While this existing work enables reasoning about the behaviour of meso- and micro-level designs of agents and agent-based systems, it does not support the stepwise development of these systems from macro-level descriptions. Our work aims to fill this gap. It uses the common action system model to specify both the behaviour of MAS and the autonomous behaviour of agents based on their local view of the environment through the messages they have received. As we clarified in Section 3.3, our model of agents captures the defining characteristics of **autonomy**, **locality** and **decentralisation**.

The application of MAZE to a substantial case study in this paper has validated its use on realistic MAS. The development process and proofs, however, are non-trivial and simplifying these is seen as important for the wider use of MAZE. In other work on the verification of concurrent data structures, Derrick et al. [29] have shown how similar proofs can be reduced to be both *thread-local* (dealing with one concurrent entity at a time) and *step-local* (dealing with one operation of a concurrent entity at a time). This results in many proof obligations, but each of them relatively straightforward to discharge. It is envisaged that a similar approach could be adopted for MAZE.

Related to our work are other approaches employing Z and Object-Z. A formal agent framework using Z is proposed by d'Inverno and Luck in [30]. The framework captures the autonomous local behaviour of an agent and inter-agent interactions. Hilaire et al. [31] propose a prototyping approach for specifying multi-agent systems. It employs a composition of Object-Z and statecharts for formalising the interaction patterns based on agent roles. Unlike these formal specifications, our approach focusses on providing a formal development framework to guarantee macro-level properties of MAS be achieved by micro-level behaviours and interactions of autonomous agents. Aştefănoaei and de Boer [32] define a notion of refinement for BDI agents. However, abstract and concrete specifications are not in the same notation. Therefore, their approach allows only a single refinement step from an abstract to a concrete representation of an agent, not the incremental development of an agent.

The Event-B formalism also advocates top-down development of software systems using refinement techniques [21, 33]. The simulation rules for action refinement in this paper are inspired by the rules in Event-B. Since publication of our first paper on MAZE [11], a similar approach based on Event-B has been proposed [34]. This paper describes a formal development approach for achieving desired system-level properties by cooperative behaviour of 'foraging ants'. The development begins from an abstract macro-level

specification and, for each refinement step, the specification provides finer and more detailed mechanisms of the agents' local behaviour and introduces cooperation. Unlike our work, however, this paper does not use the concepts of the macro, meso and micro levels from agent-based software engineering to provide a generic development approach. We believe this helps the designer to separate the macro-level concerns of system functionality, the meso-level concerns of agent interaction and the micro-level concerns of local agent behaviour by focussing on one of these sets of concerns at a time. Additionally, our syntactic conventions at the micro level allow us to readily abstract from low-level mechanisms dealing with asynchronous communication and timing constraints. Such low-level mechanisms are central to the operation of asynchronous MAS [22, 23] and without our conventions their specification would need to be intermingled with those of the particular system under development resulting in less readable specifications.

6. Conclusion

This paper has presented MAZE, an extension of Object-Z [10] for the specification and development of multi-agent systems, and its application to a swarm robotic self-assembly algorithm. MAZE supports the development of multi-agent systems at three distinct levels of abstraction proposed by researchers in the agent-oriented software engineering community. Macro-level specifications capture global system properties, meso-level specifications additionally include agent interactions and interaction paradigms, and micro-level specifications focus on the local behaviours of individual agents. To ensure the three levels are consistent, MAZE employs a notion of action refinement based on that of action systems [12], with practical proof rules inspired by those of Event-B [21]. To facilitate specification at the micro level, MAZE also includes a number of syntactic constructs. These enable the specifier to abstract from the low-level details of asynchronous communication and timing mechanisms common in such systems [22].

Future work on MAZE will focus on simplifying the verification process and providing tool support. One possibility for the latter is translation from the MAZE notation to Event-B allowing the use of the Rodin toolkit [35]. Other interesting directions include direct support for notions of spatial location and movement in 2D and 3D space, linking MAZE to high-level BDI models of agents, and introducing time and probabilistic notions to MAZE.

Acknowledgements This work was supported by Australian Research Council (ARC) Discovery Grant DP110101211 and National Science Foundation of China NSFC 61402176.

References

References

[1] M. Wooldridge, An Introduction to MultiAgent Systems, 2nd Edition, Wiley, 2009.
[2] A. Rao, M. Georgeff, BDI agents: From theory to practice, in: International Conference on Multi-Agent Systems (ICMAS-95), 1995, pp. 312–319.
[3] F. Dressler, Self-organization in sensor and actor networks, Wiley, 2007.
[4] M. Gerla, T. Kwon, G. Pei, On demand routing in large ad hoc wireless networks with passive clustering, in: Wireless Communications and Networking Conference (WCNC 2000), Vol. **1**, 2000, pp. 100–105.

[5] G. Rzevski, P. Skobelev, Managing Complexity, WIT Press, 2014.
[6] F. Zambonelli, A. Omicini, Challenges and research directions in agent-oriented software engineering, Autonomous Agents and Multi-Agent Systems 9 (3) (2004) 253–283.
[7] G. Smith, J. Sanders, Formal development of self-organising systems, in: J. González Nieto, W. Reif, G. Wang, J. Indulska (Eds.), International Conference on Autonomic and Trusted Computing (ATC 2009), Vol. 5586 of LNCS, Springer-Verlag, 2009, pp. 90–104.
[8] S. Eder, G. Smith, An approach to formal verification of free-flight separation, in: Self-Adaptive and Self-Organizing Systems Workshop (SASOW 2010), IEEE Computer Society Press, 2010, pp. 166–171.
[9] G. Smith, K. Winter, Incremental development of multi-agent systems in Object-Z, in: Software Engineering Workshop (SEW-35), IEEE Computer Society Press, 2012.
[10] G. Smith, The Object-Z Specification Language, Kluwer Academic Publishers, 2000.
[11] G. Smith, Q. Li, MAZE: An extension of Object-Z for multi-agent systems, in: Y. A. Ameur, K.-D. Schewe (Eds.), 4th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ 2014), Vol. 8477 of LNCS, Springer-Verlag, 2014, pp. 72–85.
[12] R. Back, J. von Wright, Trace refinement of action systems, in: B. Jonsson, J. Parrow (Eds.), 5th International Conference on Concurrency Theory (CONCUR '94), Vol. 836 of LNCS, Springer-Verlag, 1994, pp. 367–384.
[13] J. Spivey, The Z Notation: a reference manual, 2nd Edition, Prentice-Hall, 1992.
[14] K. Støy, Using cellular automata and gradients to control self-reconfiguration, Robotics and Autonomous Systems 54 (2006) 135–141.
[15] K. Støy, R. Nagpal, Self-reconfiguration using directed growth, in: International Symposium on Distributed Autonomous Robotic Systems (DARS 6), Springer-Verlag, 2007, pp. 3–12.
[16] C. Jones, M. Matarić, From local to global behavior in intelligent self-asssembly, in: IEEE International Conference on Robotics and Automation (ICRA'03), IEEE, 2003, pp. 721–726.
[17] E. Klavins, R. Ghrist, D. Lipsky, A grammatical approach to self-organizing robotic systems, IEEE Transactions on Automatic Control 51 (6) (2006) 949–962.
[18] A. Christensen, R. O'Grady, M. Dorigo, SWARMORPH-script: A language for arbitrary morphology generation in self-assembling robots, Swarm Intelligence 2 (2-4) (2008) 143–165.
[19] W. Lui, A. Winfield, Autonomous morphogenesis in self-assembling robots using IR-based sensing and local communications, in: M. Dorigo, M. Birattari, G. D. Caro, R. Doursat, A. Engelbrecht, D. Floreano, L. Gambardella, R. Groß, E. Şahin, H. Sayama, T. Stützle (Eds.), International Conference on Swarm Intelligence (ANTS 2010), Vol. 6234 of Lecture Notes in Computer Science, Springer-Verlag, 2010, pp. 107–118.
[20] J. Derrick, E. Boiten, Refinement in Z and Object-Z, Foundations and Advanced Applications, 2nd Edition, Springer-Verlag, 2014.
[21] J.-R. Abrial, Modelling in Event-B, Cambridge University Press, 2010.
[22] C. Chou, I. Cidon, I. Gopal, S. Zaks, Synchronizing asynchronous bounded delay networks, IEEE Trans. Communications 38 (2) (1990) 144–147.
[23] Q. Li, G. Smith, Using bounded fairness to specify and verify ordered asynchronous multi-agent systems, in: International Conference on Engineering of Complex Computer Systems (ICECCS 2013), IEEE Computer Society Press, 2013, pp. 111–120.
[24] K. Hindriks, J.-J. Meyer, Toward a programming theory for rational agents, Autonomous Agents and Multi-Agent Systems 19 (1) (2009) 4–29.
[25] M. Fisher, Agent deliberation in an executable temporal framework, Journal of Applied Logic 9 (4) (2011) 223 – 238, special Issue on Logics for Intelligent Agents and Multi-Agent Systems.
[26] M. Wooldridge, N. R. Jennings, Intelligent agents: Theory and practice, Knowledge Engineering Review **10** (1995) 115–152.
[27] A. Rao, M. Georgeff, Decision procedures for BDI logics, Journal of Logic and Computation 8 (3) (1998) 293–342.
[28] J.-J. C. Meyer, Logics for intelligent agents and multi-agent systems, in: D. Gabbay, J. Woods (Eds.), Handbook of the History of Logic, Elsevier, 2014, pp. 629–658.
[29] J. Derrick, G. Schellhorn, H. Wehrheim, Mechanically verified proof obligations for linearizability, ACM Trans. Program. Lang. Syst. 33 (1) (2011) 4.
[30] M. d'Inverno, M. Luck, Development and application of a formal agent framework, in: First IEEE International Conference on Formal Engineering Methods (ICFEM '97), IEEE Computer Society, 1997.
[31] V. Hilaire, A. Koukam, P. Gruer, J.-P. Müller, Formal specification and prototyping of multi-agent systems, in: A. Omicini, R. Tolksdorf, F. Zambonelli (Eds.), Engineering Societies in the Agents

World, Vol. 1972 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2000, pp. 114–127.
[32] L. Aştefănoaei, F. de Boer, The refinement of multi-agent systems, in: M. Dastani, K. Hindriks, J.-J. Meyer (Eds.), Specification and Verification of Multi-agent Systems, Springer-Verlag, 2010, Ch. 2, pp. 35–65.
[33] W. Su, J.-R. Abrial, R. Huang, H. Zhu, From requirements to development: Methodology and example, in: 13th International Conference on Formal Engineering Methods, (ICFEM 2011), Springer, 2011, pp. 437–455.
[34] L. Laibinis, E. Troubitsyna, Z. Graja, F. Migeon, A. Hadj Kacem, Formal modelling and verification of cooperative ant behaviour in Event-B, in: D. Giannakopoulou, G. Salaün (Eds.), Software Engineering and Formal Methods (SEFM 2014), Vol. 8702 of Lecture Notes in Computer Science, Springer International Publishing, 2014, pp. 363–377.
[35] J.-R. Abrial, M. Butler, S. Hallerstede, T. Hoang, F. Mehta, L. Voisin, Rodin: an open toolset for modelling and reasoning in Event-B, International Journal on Software Tools for Technology Transfer 12 (6) (2010) 447–466.

Appendix

This appendix gives the definitions of the rest of the actions in the micro-level specification of the swarm robotic self-assembly system in Section 4.

The meso-level action $Join_4$ is simulated by a join process consisting of a series of micro-level actions: *Request*, *SendPosition* and *Join*. *Request* is as defined in Section 4. *SendPosition* models a recruiter receiving a request message and responding to the moving atom with a *target* message indicating a vacant target position.

SendPosition

$b : Atom$
$p : Position$

$\mathbf{receive}(request, b)$
$joined$
$p \in \{p : target \mid (pos, p) \in nb\} \setminus filled$
$\mathbf{send}(target(target, p), b)$

Join

$\Delta(grad, next, target, pos, joined)$
$t : \mathbb{F}\, Position$
$p : Position$

$\mathbf{receive}(target(t, p), b)$
$grad' = \infty \land next' = \varnothing$
$target' = t \land pos' = p \land joined'$
$\mathbf{send}(moving)$

When the atom receives the *target* message, it can perform action *Join* to join the target position. A synchronised system action $Join < [] \; a : \mathrm{dom}\, atoms \bullet a.Join >$ is also defined to change the topology of the system when the join succeeds.

$Join < [] \; a : \mathrm{dom}\, atoms \bullet a.Join >$

$\Delta(position)$
$p : Position$

$p \notin \mathrm{ran}\, position \lor p = position(a)$
$p \notin \mathrm{ran}\, position \Rightarrow position' = position \oplus \{a \mapsto p\}$
$p = position(a) \Rightarrow position' = position$

The proof that $Join_5$ simulates $Join_4$ is similar with the proof that $Move_5$ simulates $Move_4$ given in Section 4. As with the *Move* case, a deadlock can happen when an atom receives a *target* message but is unable to move to the target position because it is

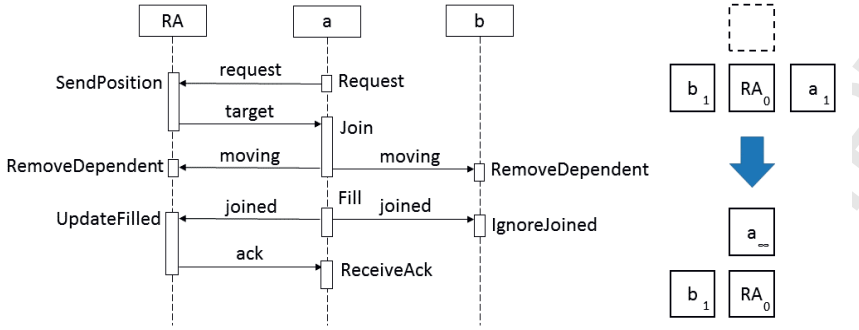


Figure 8: Sequence diagram for joining process.

occupied by another atom. In this case, the atom needs to ignore the position and send another request. This is modelled by an additional stuttering agent action and system synchronisation action *CancelJoin*.

CancelJoin

$b : Atom$
$t : \mathbb{F}\, Position$
$p : Position$

$\mathbf{receive}(target(t, p), b)$

CancelJoin $< a : atoms \bullet a.CancelJoin >$

$p : Position$

$p \in \operatorname{ran} position \land p \neq position(a)$

Figure 8 shows the sequence diagram of the joining process where an atom a joins the dashed target position. The process after action *Join* is described below.

In order to simulate the meso-level action *DissipateGrad*, we define two micro-level actions *RemoveGrad* and *Dissipate* to cover the two situations specified in *DissipateGrad*.

RemoveGrad models an atom with $grad = 0$ and all its neighbouring target positions filled setting its gradient value to ∞ and broadcasting a *dissipate* message. *Dissipate* models an atom with $b \in next$ receiving a *dissipate* message from b, setting its gradient value to ∞ and propagating the *dissipate* message to its neighbours; otherwise, it does nothing but absorb this redundant *dissipate* message.

RemoveGrad

$\Delta(grad)$

$grad = 0$
$filled = \{p : target \mid (pos, p) \in nb\}$
$grad' = \infty$
$\mathbf{send}(dissipate)$

Dissipate

$\Delta(grad, next)$
$b : Atom$

$\mathbf{receive}(dissipate, b)$
$b \notin next \Rightarrow grad' = grad \land next' = next$
$b \in next \Rightarrow$
$\quad grad' = \infty \land next' = next \setminus \{b\} \land$
$\quad \mathbf{send}(dissipate)$

According to the meso-level action *DissipateGrad*, a recruiter will dissipate its gradient when all neighbouring target positions are filled so that unplaced atoms can turn to

another recruiter. However, at the micro level, a recruiter does not realise the recruiting target is filled until the joined atom sends it a notification. Without such a notification, the recruiter could keep referring the already filled target position to other atoms. Therefore, we define the following stuttering actions to maintain the *filled* set of a placed atom.

Fill models an atom which has joined the target position broadcasting a *joined* message to notify its new neighbours that the target position is filled. The variable *notified* is used to ensure that the *joined* message is only sent once. *UpdateFilled* models the atom which receives the *joined* message updating its *filled* set and returning an *ack* message with its position to let the new joined atom update its own *filled* set.

Fill

$\Delta(\mathit{notified})$

$\mathit{joined} \wedge \neg \mathit{notified}$
$\mathit{notified}'$
$\mathbf{send}(\mathit{joined}(\mathit{pos}))$

UpdateFilled

$\Delta(\mathit{filled})$
$b : \mathit{Atom}$
$p : \mathit{Position}$

$\mathbf{receive}(\mathit{joined}(p), b)$
joined
$\mathit{filled}' = \mathit{filled} \cup \{p\}$
$\mathbf{send}(\mathit{ack}(\mathit{pos}), b)$

IgnoreJoined models an atom which is not placed in a target position ignoring a *joined* message. *ReceiveAck* models an atom receiving an *ack* message from one of its neighbours and updating its *filled* variable with the position. Note that *ack* messages are only sent in response to a *joined* message and hence will only be received by atoms which have joined the target.

IgnoreJoined

$b : \mathit{Atom}$
$p : \mathit{Position}$

$\mathbf{receive}(\mathit{joined}(p), b)$
$\neg \mathit{joined}$

ReceiveAck

$\Delta(\mathit{filled})$
$b : \mathit{Atom}$
$p : \mathit{Position}$

$\mathbf{receive}(\mathit{ack}(p), b)$
$\mathit{filled}' = \mathit{filled} \cup \{p\}$

With the above stuttering actions, we can now prove that *atoms.RemoveGrad* simulates $\mathit{DissipateGrad}_4$ for the first situation and *atoms.Dissipate* simulates $\mathit{DissipateGrad}_4$ for the second situation.

According to *SetGrad*, the guard $\mathit{grad} = 0$ of *RemoveGrad* implies that all *joined* messages are handled so that the current *filled* set records all filled neighbouring target positions. Therefore, we have $R4 \Rightarrow (\mathrm{pre}\ \mathit{atoms.RemoveGrad} \Rightarrow \mathrm{pre}\ \mathit{DissipateGrad})$. It is obvious that the effect of *RemoveGrad* implies *DissipateGrad*. In summary, the simulation holds.

For an atom a performing $a.\mathit{Dissipate}$, its guard $b \in \mathit{next}$ implies $a.\mathit{grad} = b.\mathit{grad} + 1$ which is consistent with the second branch of pre *DissipateGrad*. Hence we can complete the proof with trivial deductions.